# Services et protocoles pour l'exécution fiable d'applications distribuées dans les grilles de calcul

Thomas Ropars

N° d'ordre : 3992 ANNÉE 2009



**THÈSE / UNIVERSITÉ DE RENNES 1**
*sous le sceau de l'Université Européenne de Bretagne*

pour le grade de
**DOCTEUR DE L'UNIVERSITÉ DE RENNES 1**

*Mention : Informatique*
**Ecole doctorale MATISSE**

présentée par

**Thomas Ropars**

préparée à l'unité de recherche 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

---

# Services et protocoles pour l'exécution fiable d'applications distribuées dans les grilles de calcul

**Thèse soutenue à Rennes**
**le 11 décembre 2009**

devant le jury composé de :

**Franck Cappello**
Directeur de Recherche, INRIA - University of Illinois / Rapporteur

**Frédéric Desprez**
Directeur de Recherche, INRIA Grenoble - Rhône-Alpes / Rapporteur

**Anne-Marie Kermarrec**
Directrice de Recherche, INRIA Rennes - Bretagne Atlantique / Examinateur

**Pierre Sens**
Professeur, Université Paris 6
Examinateur

**André Schiper**
Professeur, EPFL
Examinateur

**Christine Morin**
Directrice de Recherche, INRIA Rennes - Bretagne Atlantique / Directeur de thèse

# Remerciements

Cette thèse est le résultat de trois années de travail, mais aussi et surtout le résultat de nombreuses rencontres enrichissantes.

Mes remerciements vont tout d'abord aux membres du jury qui m'ont fait l'honneur de prendre le temps d'évaluer mes travaux. Merci à Franck Cappello et Frédéric Desprez d'avoir accepter le rôle de rapporteur. Merci pour la pertinence de leurs commentaires et pour avoir fait l'effort de respecter les délais imposés en dépit des retards que j'avais accumulé. Merci à Pierre Sens d'avoir accepté de présider mon jury. Merci enfin à Anne-Marie Kermarrec et André Schiper de l'intérêt qu'ils ont porté à mes travaux.

Je remercie Thierry Priol de m'avoir donner la possibilité d'effectuer mes travaux au sein de l'équipe-projet PARIS.

Merci Christine pour ton encadrement depuis maintenant près de 4 ans. Je te remercie de la confiance que tu m'as accordée, de ton soutien dans les moments de doute et de la liberté qui tu m'as offerte pour mener mes travaux tels que je l'entendais.

Merci Yvon, David et Peter pour ces discussions enrichissantes lors des pauses café. Merci Maryse, toujours là pour nous simplifier la vie.

Merci Adrien, André, Boris, Cyril, Jérôme, Julien, Hinde, Landry, Loïc, Marko, Mathieu, Matthieu, Oscar, Pascal, Pierre, Skalp, Sylvain et Xuanhua pour tous les bons moments passés ensemble. Merci Raúl pour les discussions du petit déjeuné pour bien commencer la journée.

Merci Aurélien, Catalin et John pour ces collaborations fructueuses. J'espère qu'il y en aura d'autres dans le futur. Merci Emmanuel de m'avoir mis le pied à l'étrier. Travailler avec toi fut très agréable. Merci Rajib, Sébastien et Stefania d'avoir accepté de partager une partie de l'aventure avec moi. J'ai pris beaucoup de plaisir à travailler avec vous. Je vous souhaite le meilleur pour la suite.

Mes remerciements s'adressent enfin aux personnes qui comptent le plus pour moi. Merci Maman, merci Ludo pour votre soutien et votre aide au cours de toutes ces années. Je termine ces remerciements en ayant une pensée pour mon père et mes grands-parents.

# Table des matières

# Table des figures

# Liste des définitions

# Introduction

Face à un problème aussi complexe à résoudre que prévoir quel temps il fera en Bretagne demain, les hommes n'ont d'autres recours que l'informatique. La communauté scientifique se sert de l'informatique comme d'un outil pour simuler, analyser et comprendre des phénomènes complexes. L'analyse des phénomènes météorologiques en est un exemple. Étudier la résistance d'une centrale nucléaire à un tremblement de terre en est un autre.

Étudier de tels problèmes requiert des capacités de calcul qu'un ordinateur personnel ne peut fournir. Les systèmes distribués permettent alors d'agréger les puissances de calcul de plusieurs ressources informatiques pour répondre à ces besoins. Plus les capacités de calcul disponibles sont importantes, plus les quantités de données analysées peuvent être importantes, plus les calculs peuvent être précis et plus les phénomènes peuvent être finement analysés. Dans ce contexte, les grilles de calcul, dont le principe est d'agréger un grand nombre de ressources de calcul réparties géographiquement, sont une solution attrayante car elles permettent à des entreprises et des universités de mettre en commun leur puissance de calcul.

Si les grilles de calcul représentent une solution attractive pour répondre aux besoins en capacité de calcul de la communauté scientifique, leur utilisation est complexe même pour des spécialistes. Une grille de calcul peut regrouper plusieurs dizaines de milliers de ressources. Des ressources peuvent y être ajoutées ou retirées à tout moment. De plus, étant donné ses dimensions, la probabilité de défaillances y est élevée. Enfin, par nature, les ressources composant une grille de calcul sont hétérogènes au niveau matériel et logiciel. C'est pourquoi de nombreux travaux visent à fournir aux utilisateurs des services simplifiant l'utilisation des grilles de calcul, de la soumission des applications à l'obtention des résultats.

Les utilisateurs d'une grille de calcul doivent obtenir le résultat des applications qu'ils soumettent. Si tel n'est pas le cas, tous les investissements consentis pour la mise en place d'un environnement d'exécution tel qu'une grille de calcul sont vains. Cependant assurer la bonne terminaison d'applications distribuées sur un grand nombre de ressources volatiles, et dont la durée d'exécution peut dépasser plusieurs jours voire plusieurs semaines, est un réel défi.

## Objectifs de la thèse

Les travaux présentés dans cette thèse visent à **assurer l'exécution fiable d'applications distribuées**, potentiellement de grande taille, dans les grilles de calcul.

Pour atteindre cet objectif, doivent être traités les problèmes liés à la volatilité des ressources de la grille, et en particulier les conséquences des défaillances. C'est pourquoi cette thèse s'intéresse à la problématique de la tolérance aux fautes dans les systèmes distribués de grande taille. La tolérance aux fautes, et en particulier la tolérance aux fautes dans les systèmes distribués, est un domaine très actif de la recherche en informatique.

Les solutions de tolérance aux fautes proposées dans cette thèse doivent être adaptées à **la taille**, **la volatilité** et **l'hétérogénéité** des grilles de calcul.

Pour répondre aux attentes des utilisateurs au-delà du besoin d'assurer l'obtention des résultats, les solutions de tolérance aux fautes proposées doivent contribuer à faire des grilles de calcul un environnement d'exécution **simple à utiliser** et **performant**. Les mécanismes de tolérance aux fautes doivent simplifier l'utilisation des grilles de calcul en rendant les défaillances et leur gestion transparentes pour les utilisateurs. De plus, ils doivent avoir un impact limité sur les performances des applications.

## Contributions

Les contributions présentées dans cette thèse se déclinent en trois axes.

**Un service de recouvrement arrière pour applications distribuées.** Ce service, nommé XtreemGCP et proposé dans le contexte du projet européen XtreemOS, assure la terminaison des applications s'exécutant sur la grille en dépit des défaillances. XtreemGCP est capable, en interagissant avec les autres services de XtreemOS, de redémarrer automatiquement les applications ayant subi une défaillance. Fondé sur une architecture passant à l'échelle, XtreemGCP est conçu de manière générique pour pouvoir profiter des nombreux travaux sur la tolérance aux fautes. Ainsi il est capable d'exploiter des mécanismes de tolérance aux fautes directement inclus dans l'application ou fournies par des bibliothèques tolérantes aux fautes, telles que les bibliothèques MPI. De plus, nous proposons l'utilisation d'une interface générique pour les mécanismes de sauvegarde de points de reprise de processus, offrant la possibilité d'intégrer au service différents outils, et permettant donc de prendre en charge des applications distribuées sur des ressources hétérogènes. Enfin XtreemGCP permet la mise en œuvre de différents protocoles de recouvrement arrière de manière transparente pour les applications. Nous décrivons dans ce document la mise en œuvre d'un protocole de sauvegarde de points de reprise coordonnés. Toutes ces solutions permettent à l'utilisateur ou au programmeur de l'application de choisir les mécanismes de tolérance aux fautes les mieux adaptés à l'application.

**Un cadre pour la mise en œuvre de services hautement disponibles et auto-réparants.** Pour assurer la disponibilité de services de grille tels que XtreemGCP en dépit de volatilité des ressources de la grille et notamment des défaillances, nous proposons Semias, un cadre pour la mise en œuvre de services hautement disponibles et auto-réparants. Fondé sur l'utilisation combinée de duplication active et d'un réseau logique structuré, Semias rend les défaillances et reconfigurations de services dupliqués totalement transparentes pour les utilisateurs des services. Semias est capable de prendre en charge un grand nombre de services et d'exploiter l'ensemble des ressources de la grille pour exécuter les services dupliqués. Les propriétés d'auto-réparation offertes par Semias assure la disponibilité des services dans un environnement d'exécution dynamique, sans intervention humaine. Un prototype de Semias a été mis en œuvre et utilisé pour rendre le service de gestion d'applications de l'intergiciel de grille Vigne hautement disponible. Les modifications nécessaires pour intégrer un service existant à Semias sont légères. Les évaluations menées sur la plate-forme Grid'5000 ont montré l'efficacité des mécanismes d'auto-réparation de Semias.

**Un protocole de recouvrement arrière pour applications à échange de messages de grande taille.** L'échange de messages est un paradigme de programmation

très couramment utilisé pour les applications de calcul scientifique , et plus spécifiquement pour les applications parallèles. Nous avons conçu et mis en œuvre O2P, un protocole de recouvrement arrière fondé sur l'enregistrement de messages optimiste actif, offrant une solution de tolérance aux fautes adaptée aux applications à échange de messages de grande taille. L'enregistrement optimiste actif limite le surcoût des mécanismes de recouvrement arrière sur les performances d'une application lors d'une exécution sans défaillance. De plus, il permet à O2P de mieux passer à l'échelle que les protocoles à enregistrement de messages optimistes existants. Nous apportons la preuve que O2P est capable de supporter la défaillance simultanée de plusieurs processus. Pour améliorer le passage à l'échelle des protocoles à enregistrement de messages, nous proposons de plus une solution de gestion distribuée de l'enregistrement des messages. Nous avons mis en œuvre O2P dans la bibliothèque Open MPI et évalué notre prototype sur la plate-forme Grid'5000. Ces évaluations démontrent l'intérêt de l'enregistrement de messages optimiste actif. Elles montrent de plus, le meilleur passage à l'échelle de notre solution distribuée pour l'enregistrement des messages par rapport à la solution centralisée traditionnellement considérée.

L'exploitation combinée de ces trois contributions permet d'atteindre nos objectifs. XtreemGCP peut utiliser O2P pour assurer l'exécution fiable d'applications de grande taille dans une grille de calcul. Semias permet d'assurer la disponibilité de XtreemGCP en dépit de la volatilité des ressources de la grille.

Ces travaux sont menés pour partie dans le contexte du projet européen XtreemOS [204] et pour partie dans celui de l'intergiciel de grille Vigne [159], développé au sein de l'équipe-projet PARIS. Les évaluations sont menées sur la plate-forme expérimentale Grid'5000.

## Organisation du document

Ce document est organisé de la façon suivante. Le chapitre 1 introduit la problématique de la gestion des défaillances dans les grilles de calcul et les enjeux qui y sont associés. Nous concluons ce chapitre en définissant le modèle de faute que nous associons aux grilles de calcul et que nous utilisons dans la suite du document. Le chapitre 2 décrit notre approche pour assurer l'exécution fiable d'applications dans les grilles de calcul et introduit les trois axes de recherche que nous avons suivis, auxquels nous consacrons chacun un chapitre.

Le chapitre 3 décrit la conception du service XtreemGCP, un service chargé d'assurer la terminaison des applications s'exécutant sur la grille en utilisant des techniques de recouvrement arrière. Le chapitre 4 présente Semias, notre cadre pour la mise en œuvre de services hautement disponibles et auto-réparants dans la grille. Le chapitre 5 présente notre protocole à enregistrement de message optimiste actif, O2P, visant les applications à échange de messages de grande taille. Dans chacun de ces trois chapitres, nous commençons par positionner notre contribution par rapport à l'état de l'art avant de présenter les propositions et leur évaluation. Une évaluation de Semias et du protocole O2P a été menée sur la plate-forme expérimentale Grid'5000.

Nous concluons par un chapitre dressant un bilan de nos contributions et présentant les perspectives ouvertes par nos travaux.

# Chapitre 1

# Les défaillances dans les grilles de calcul

Dans ce chapitre, nous présentons la problématique des défaillances dans les grilles de calcul et les enjeux associés.

Commençons par introduire quelques notions fondamentales de l'informatique. L'informatique est fondée sur l'exécution d'une suite d'instructions par une unité de traitement. Cette suite d'instructions est obtenue par la compilation d'un programme informatique. Ce programme est un ensemble d'actions et de comportements décrit par un programmeur à l'aide d'un langage de programmation.

**Définition 1.1 (Unité de traitement)** *Une unité de traitement (*central processing unit (CPU)*) est le composant qui interprète les instructions et traite les données d'un programme informatique. Dans un processeur multi-cœurs, chaque cœur est une unité de traitement.*

La compilation transforme un ensemble d'instructions compréhensible par l'être humain en un programme exécutable, composé d'une suite d'instructions compréhensibles par l'unité de traitement. Les instructions d'un exécutable sont exécutées dans le contexte d'un processus.

**Définition 1.2 (Processus)** *Un processus est le contexte d'exécution d'un programme. Dans les systèmes UNIX, est associé à chaque processus un identifiant unique, appelé* PID *(*Process ID*), des propriétés (environnement, droits,* etc*), un espace mémoire, un ensemble de registres et des moyens de communication.*

Dans un système informatique, les processus sont regroupés au sein d'applications.

**Définition 1.3 (Application)** *Une application est un ensemble de processus participant à un objectif commun.*

La nature et la fonction des applications sont diverses. Elles s'étendent du simple éditeur de texte à l'application simulant les effets d'un tsunami en passant par le jeu sur Internet gérant des millions de joueurs, *etc.* Les caractéristiques et les besoins des applications sont très variables en fonction de leur nature.

Le contexte de nos travaux est le calcul haute performance. Les grilles de calcul sont des systèmes conçus pour répondre aux besoins des applications de ce domaine. Dans ce chapitre, nous présentons les grilles de calcul et leur principales caractéristiques, mettant ainsi en évidence les besoins en tolérance aux fautes dans de tels systèmes. Dans un deuxième temps, nous décrivons les fondements de la tolérance aux défaillances dans les systèmes distribués et proposons un modèle de défaillance adapté aux grilles de calcul.

## 1.1 Les grilles de calcul

Le calcul haute performance est le domaine de l'informatique regroupant les travaux sur les applications nécessitant de grandes capacités de calcul ou manipulant de larges ensembles de données. Ces applications, souvent issues du domaine scientifique, permettent de traiter des problèmes très complexes impossibles à résoudre sans l'aide de l'informatique. La prévision météorologique, la bio-informatique ou la simulation numérique sont des exemples de domaines nécessitant de grandes capacités de calcul.

### 1.1.1 Un peu d'histoire

Pour répondre aux besoins du calcul haute performance, l'utilisation d'une seule unité de traitement est très vite devenue insuffisante. Le parallélisme, consistant à faire travailler plusieurs unités de traitement simultanément, est alors apparue comme la solution pour augmenter la puissance de calcul disponible. Des machines massivement parallèles ont été construites à partir des années 70, regroupant au sein d'une même machine physique de nombreuses unités de traitement et utilisant des architectures mémoire et des solutions de communication spécifiques.

**Définition 1.4 (Machine physique)** *Une machine physique est une unité électriquement indépendante permettant l'exécution de processus.*

La principale limite de ces machines massivement parallèles est leur coût. Les grappes de calcul sont une alternative moins coûteuse en terme d'investissements. Elles sont en général moins performantes même si à partir de 2002, elles font leur apparition dans les premières places du classement des grands centres de calcul [174].

**Définition 1.5 (Grappe de calcul)** *Une grappe de calcul connecte plusieurs machines physiques, aussi appelées serveurs, équipées d'un petit nombre d'unités de traitement au sein d'un réseau local ou d'un SAN, pour fonctionner en parallèle.*

**Définition 1.6 (Réseau local ou LAN)** *Un réseau local ou LAN (*Local Area Network*) est un réseau d'interconnexion reliant un ensemble de machines physiques d'une même organisation et proches géographiquement, c'est-à-dire situés dans la même pièce ou le même bâtiment.*

**Définition 1.7 (SAN)** *Un SAN (*System Area Network*) est un réseau dédié à l'interconnexion de machines physiques au sein d'une grappe de calcul, offrant des débits élevés et des latences faibles. Les technologies les plus connues dans ce domaine sont Myrinet [135] et Infiniband [95].*

La notion de méta-ordinateur (*metacomputer*) [176] est apparue au début des années 90 pour désigner un ordinateur virtuel composé de plusieurs machines physiques distribuées géographiquement et hétérogènes. Les méta-ordinateurs devaient remplacer les machines massivement parallèles devenues trop onéreuses.

Ce terme fut rapidement remplacé par celui de grille de calcul [75] faisant référence à l'analogie avec le réseau électrique (*power grid*). En effet de la même manière qu'une prise électrique permet de se brancher sur le réseau électrique et obtenir instantanément l'énergie dont on a besoin, on pense alors qu'un utilisateur doit pouvoir obtenir la puissance de calcul dont il a besoin en se *branchant* sur la grille de calcul au travers d'une simple prise réseau. Voici notre définition d'une grille de calcul.

**Définition 1.8 (Grille de calcul)** *Une grille de calcul (qui pourra simplement être nommé* grille *dans la suite de ce document) est un système distribué qui regroupe un très grand nombre de machines physiques hétérogènes au travers d'un WAN. Ces machines physiques peuvent appartenir à différents domaines d'administration.*

**Définition 1.9 (Système distribué)** *Un système distribué est un système composé de plusieurs machines physiques.*

**Définition 1.10 (Réseau étendu ou WAN)** *Un réseau étendu ou WAN (*Wide Area Network*) est un réseau d'interconnexion couvrant une large étendue géographique,* i.e. *un pays ou un continent. Le plus grand réseau étendu est le réseau Internet.*

**Définition 1.11 (Domaine d'administration)** *Des machines physiques appartiennent au même domaine d'administration si elles sont sous le contrôle de la même autorité administrative.*

Aujourd'hui les grilles permettent à différentes organisations de partager leur ressources de calcul pour obtenir des capacités comparables à celles des grands centres de calcul. La notion de grille est alors associée à celle d'organisation virtuelle [76].

**Définition 1.12 (Organisation virtuelle)** *Une organisation virtuelle est le résultat d'un accord entre plusieurs institutions pour partager des ressources informatiques dans un objectif commun.*

**Définition 1.13 (Ressource informatique)** *Les ressources informatiques désignent les éléments matériels permettant de stocker ou de manipuler des données. Généralement une machine physique peut fournir des unités de traitement, de la mémoire et de l'espace de stockage.*

Une organisation virtuelle a une durée de vie limitée. Les règles régissant le partage des ressources au sein de l'organisation virtuelle sont les termes de l'accord entre les institutions. Une organisation virtuelle est une subdivision de la grille regroupant des ressources et des utilisateurs, régie par un ensemble de règles de partage spécifiques.

La grille offre donc les moyens d'interconnecter et de partager des ressources distribuées et hétérogènes appartenant à différents domaines d'administration. L'organisation virtuelle offre un cadre pour définir les règles conditionnant ce partage de ressources.

Un nouveau modèle nommé *Cloud Computing* [200] a émergé au cours des dernières années. Dans ce modèle, les ressources matérielles et logicielles sont vendues par des fournisseurs via Internet. Les clients n'achètent plus directement des machines ou des logiciels mais louent des services ou des ressources auprès de ces fournisseurs. Ainsi, les clients ne payent que pour les ressources qu'ils consomment effectivement et sont déchargés du travail de maintenance. Même s'ils partagent des objectifs communs, la relation entre le concept de *Cloud Computing* et les grilles de calcul n'est pas encore clairement définie [130]. C'est pourquoi nous n'évoquerons pas le *Cloud Computing* dans la suite de ce document.

### 1.1.2 Les applications de calcul scientifique

Commençons par étudier les applications qui peuvent exploiter les ressources fournies par une grille. Ces applications sont le plus souvent issues du domaine du calcul scientifique. Dans de nombreux domaines scientifiques, les besoins en calcul sont très importants. La bio-informatique, la météorologie ou la simulation numérique sont des exemples de domaines gourmands en capacité de calcul.

**Définition 1.14 (Application distribuée)** *Une application distribuée est une application composée de plusieurs processus pouvant s'exécuter sur des machines physiques différentes.*

Pour tirer profit des ressources d'une grille, des applications distribuées sont utilisées. Exécuter plusieurs processus concurremment sur des unités de traitement différentes augmente le nombre total d'instructions exécutées par seconde. Ainsi, plus le nombre d'unités de traitement sur lesquelles s'exécute une application distribuée est grand, plus le nombre d'instructions que peut exécuter cette application en un temps donné est important. Cela signifie que pour traiter une certaine quantité de données, une application distribuée est plus rapide qu'une application s'exécutant sur une seule unité de traitement. Cette accélération est cependant limitée par les synchronisations et les communications nécessaires entre les processus d'une application distribuée.

Au-delà des considérations de performance, une application peut être distribuée de par sa nature. Par exemple, dans le cadre d'applications faisant intervenir des codes appartenant à différentes entreprises, il peut être requis par ces entreprises d'exécuter leur code sur des ressources leur appartenant pour des raisons de confidentialité.

Il existe plusieurs modèles d'applications distribuées. Voici les principaux modèles utilisés dans le domaine du calcul scientifique et des grilles :

**Sac de tâches indépendantes :** Ce sont des applications composées d'un grand nombre de tâches indépendantes. Par exemple, dans le cadre d'applications paramétriques, le même programme est exécuté plusieurs fois avec des jeux de donnée différents. Le modèle maître-travailleurs est un cas particuliers de ce type d'applications dans lequel un maître distribue les jeux de données aux travailleurs qui exécutent le même code. Ce modèle permet notamment de gérer les cas où le nombre de tâches n'est pas connu à l'avance.

**Application parallèle :** Les processus d'une application parallèle sont fortement couplés, c'est-à-dire que les interactions entre ces processus sont fréquentes. Dans ce domaine, le paradigme de programmation le plus souvent utilisé est l'échange de messages : les processus s'échangent des données et se synchronisent en s'envoyant des messages. MPI [71] et PVM [185] sont des exemples d'interfaces pour la programmation par échange de messages. Le paradigme de mémoire partagée distribuée [9, 104] est une alternative à la programmation par échange de messages. Il offre un espace d'adressage commun à des processus distribués. Si ce paradigme est plus simple à utiliser pour le programmeur, il est aussi parfois moins performant [122]. Enfin OpenMP [52] permet sur des machines multiprocesseurs à mémoire partagée une parallélisation automatique d'un programme, et notamment des boucles, en utilisant des directives de compilation à insérer dans le code du programme.

**Application distribuée faiblement couplée :** Pour ce type d'applications, les interactions entre processus sont moins fréquentes que pour des applications parallèles. Elles sont souvent fondées sur l'appel de procédures à distance. JavaRMI [196] est un exemple de canevas pour la programmation d'applications par appel de procédure à distance. GridRPC [171] est une adaptation de ce modèle visant les grilles de calcul. Le paradigme de programmation à base de composants [35, 139], d'un niveau d'abstraction plus élevé, permet de représenter une application sous la forme d'un assemblage de boîtes noires.

**Couplage de code :** Ce modèle de programmation consiste à créer une application distribué en faisant collaborer plusieurs codes, chacun d'entre eux étant en général lui-même un code parallèle. Ce modèle de programmation est particulièrement

intéressant dans le cadre d'applications faisant intervenir plusieurs disciplines. La conception d'un satellite peut, par exemple, faire intervenir des codes venus du domaine de l'optique, de la thermodynamique ou encore de la mécanique. GridCCM [145] permet de construire des applications de couplage de code à partir de composants.

**Workflow :** Un *workflow* [80] est l'enchaînement de plusieurs tâches. Chacune d'entre elles peut elle-même être une application distribuée correspondant à un des modèles décrit précédemment. Les relations entre les tâches sont le plus souvent des dépendances temporelles, *i.e.* une tâche doit utiliser les résultats d'une tâche précédente et doit donc attendre que celle-ci soit terminée pour démarrer.

Les applications de calcul scientifiques travaillent en général sur de très grandes quantités de données. C'est pourquoi, malgré les quantités de ressources que peuvent fournir les grilles de calcul, leur temps d'exécution peut être très important, c'est-à-dire de l'ordre de plusieurs jours voir de plusieurs semaines.

### 1.1.3 L'architecture des grilles de calcul

Une grille de calcul est un système distribué composé d'un très grand nombre de nœuds. Ces nœuds, pouvant appartenir à différents domaines d'administration, fournissent des ressources à la grille pour permettre l'exécution d'applications de calcul scientifique.

**Définition 1.15 (Nœud d'une grille)** *On parle de nœuds d'un système distribué par analogie avec les graphes, les liens réseaux représentant les arcs entre les nœuds. Dans le cadre des grilles, nous considérons qu'un nœud est une machine physique.*

Une grille de calcul peut être composée de différents types de nœuds. Voici quelques modèles de grille typiques dont les caractéristiques sont directement liées aux types de noeuds dont elles sont composées.

**Calcul pair à pair :** Les grilles de calcul pair à pair utilisent les ressources de machines connectées à Internet et appartenant le plus souvent à des particuliers. Elles sont fondées sur le volontariat. Le propriétaire accepte que des calculs soient réalisés sur sa machine lorsque celle-ci est allumée et qu'il ne s'en sert pas. C'est le principe de la récupération de cycles. Les grilles de calcul pair à pair peuvent être composées de millions de nœuds. L'architecture d'une grille de calcul pair à pair est illustrée par la figure 1.1. Les applications de type sac de tâches sont particulièrement bien adaptées pour ce type de grille. BOINC [10] et SETI@home [11] sont des exemples de grilles de calcul pair à pair. Une spécificité des grilles pair à pair est le nombre non négligeable d'utilisateurs malveillants.

**Réseau de stations de travail :** Dans les grandes institutions, les entreprises ou les universités, les ressources des stations de travail sont souvent sous-exploitées. Les périodes durant lesquelles ces machines sont inactivent peuvent être utilisées pour effectuer des calculs. C'est le principe des réseaux de stations de travail [120, 134]. Comme le décrit la figure 1.2, un réseau de stations de travail regroupe des machines connectées au même réseau local. À nouveau, ce sont principalement des applications de type sac de tâches qui s'exécutent sur ce type d'architecture.

**Fédération de grappes de calcul :** C'est un cas très courant dans une grille : une institution participe à une grille en mettant à disposition les ressources de grappes de calcul. Une fédération de grappes de calcul est une grille composée de grappes de calcul appartenant à différents domaines d'administration. Son architecture est

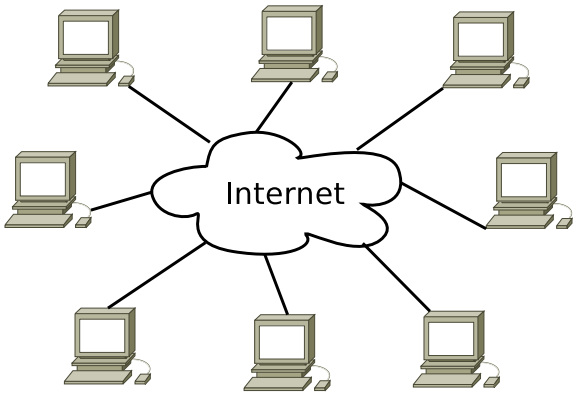



FIG. 1.1 – Grille de calcul pair à pair

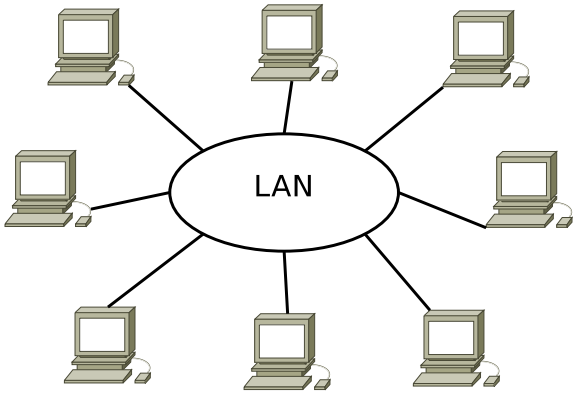



FIG. 1.2 – Réseau de stations de travail

hiérarchique comme l'illustre la figure 1.3. Des grilles de production comme SweGrid [186] ou des grilles expérimentales comme Grid'5000 [24], TeraGrid [144], ou PRAGMA [2] sont des fédérations de grappes. Ce type d'architecture est particulièrement bien adapté pour les applications de couplage de code, chaque partie de l'application pouvant être exécutée exclusivement sur une grappe de calcul.

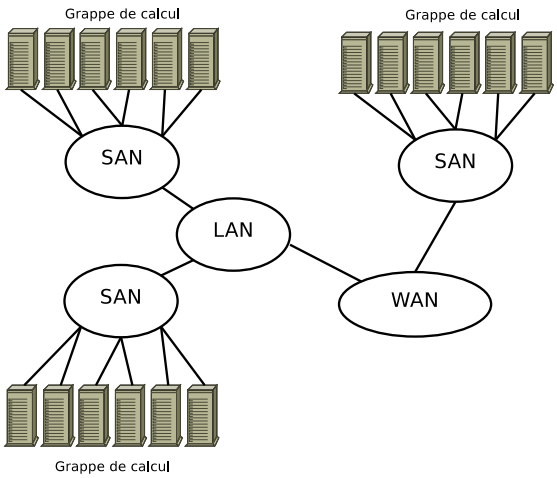



FIG. 1.3 – Fédération de grappes de calcul. Deux grappes appartenant au même domaine d'administration sont connectées par un LAN. Une troisième grappe, appartenant à un domaine d'administration différent, est connecté aux deux premières par un WAN.

**Grille hétérogène :** Une grille hétérogène peut aussi bien intégrer des stations de travail que des grappes de calcul ou des machines massivement parallèles. Ce modèle, illustré par la figure 1.4, présente comme les fédérations de grappes de calcul une architecture hiérarchique. Certaines grilles peuvent combiner des grappes de calcul et des machines massivement parallèles. C'est par exemple le cas pour NorGrid [137] ou DEISA [58]. PlanetLab [147] est une grille dont l'architecture est proche des grilles de calcul pair à pair car les nœuds la composant sont des stations de travail interconnectées via internet. Cependant ces nœuds sont des machines dédiées fournies par des institutions, ce qui limite la volatilité ainsi que les risques de fraude. EGEE [107] est un autre exemple de grille hétérogène regroupant plus de 68000 unités de traitement.

Dans le cadre de nos travaux, nous considérons un modèle de grille hétérogène car il est le plus générique.

Les grilles de données sont un autre type de grille, spécialisées dans le stockage de grands volumes de données. Par exemple LCG [83] sert à gérer les données fournies par l'accélérateur de particules du CERN [45]. Les problématiques y sont différentes de celles rencontrées dans les types de grilles que nous avons précédemment décrits, puisque le but n'est pas l'exécution d'applications. Nos travaux se concentrent sur les grilles de calcul, ces grilles pouvant intégrer en plus des ressources de stockage de données.

### 1.1.4 Les défis liés à l'utilisation des grilles de calcul

Les défis liés à l'utilisation des grilles de calcul sont nombreux. Ils sont liés à la taille de celles-ci, leur hétérogénéité ou encore leur nombre d'utilisateurs. Nous présentons maintenant ces principaux défis.

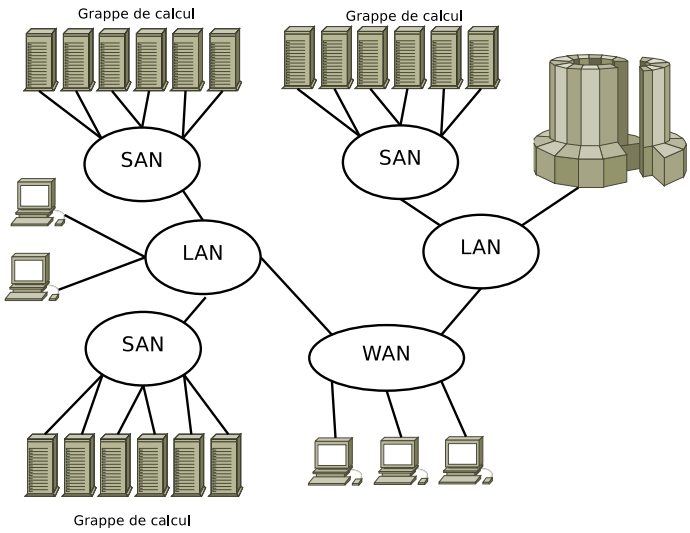



Fig. 1.4 – Grille hétérogène

**Une infrastructure de grande taille.** Une grille est composée d'un très grand nombre de nœuds. Nous pouvons envisager des grilles composées de plusieurs dizaines de milliers de nœuds. Pouvoir supporter un très grand nombre de nœuds et offrir la qualité de service requise par les applications quelque soit le nombre de nœuds considéré sont des enjeux majeurs. Il faut donc que les solutions utilisées pour l'exploitation des ressources d'une grille soient passent à l'échelle.

**Des ressources hétérogènes.** Comme nous l'avons vu précédemment, l'hétérogénéité se manifeste tout d'abord au niveau matériel. Des machines physiques de différents types peuvent être intégrées au sein de la même grille. Les réseaux sont eux aussi hétérogènes. Les latences ne sont pas les mêmes entre deux nœuds d'une même grappe de calcul reliés par un SAN et deux nœuds situés chacun à un bout de la terre et reliés par un WAN. Cette hétérogénéité doit être prise en compte pour une utilisation optimale des ressources de la grille. Cette hétérogénéité est aussi présente au niveau logiciel. Il est possible que différents systèmes d'exploitation soient utilisés au sein de la même grille et il est très peu probable que tous les nœuds d'une grille fournissent les mêmes ressources logicielles. Cette hétérogénéité logicielle peut au moins être en parti résolu par l'utilisation de technologies de virtualisation [17, 203]. Il est ainsi possible de sélectionner le système d'exploitation qui sera utilisé dans une machine virtuelle. C'est pourquoi dans la suite de ce document, nous ne considérons que des systèmes d'exploitation de type Linux.

**Des ressources volatiles.** La volatilité des nœuds d'une grille a plusieurs causes. Celle-ci peut être due à des déconnections volontaires, par exemple dans le cas d'une station travail qui est déconnectée pour être utilisée par son propriétaire. Certaines machines peuvent aussi être arrêtées lors d'opérations de maintenance. De la même manière de nouveaux nœuds peuvent à tout instant être mis à la disposition de la grille. Enfin étant donné le nombre de nœuds composant une grille de calcul, la probabilité de défaillance d'un de ces nœuds est très élevée. La défaillance de liens d'interconnexion entraîne aussi la perte des nœuds en dépendant. Ces déconnections de nœuds, subies ou volontaires, peuvent affectées la bonne terminaison des applications s'exécutant sur la grille. Nous abordons plus en détail cet aspect dans la section 1.1.6.

**Un grand nombre d'utilisateurs et d'applications.** Le nombre d'utilisateurs d'une grille est en général très grand. Par conséquent, le nombre d'applications qui s'exécutent sur une grille, parfois au même moment, est lui aussi très grand. Gérer ce grand nombre d'utilisateurs et d'applications est une tâche complexe. De plus, ce grand nombre d'utilisateurs provenant d'institutions différentes est un défi pour la sécurité. Ce problème étant transversal à tout les services offerts sur une grille, nous ne l'abordons pas dans ce document.

### 1.1.5 Les systèmes de grille

Un scénario simple d'exécution d'une application sur une grille peut se décomposer en plusieurs phases. Tout d'abord l'utilisateur doit trouver les ressources dont il a besoin pour exécuter son application. Il doit ensuite transférer le code exécutable de son application, ainsi que les fichiers de données si besoin, vers les nœuds sélectionnés et démarrer son application. Enfin il doit récupérer les résultats une fois l'exécution terminée. Toutes ces tâches paraissent simples. Cependant, dans le contexte aussi complexe que celui d'une grille de calcul, elles peuvent se révéler très difficiles. C'est pourquoi un système de grille est utilisé pour fournir aux utilisateurs un ensemble de services en simplifiant l'utilisation.

**Définition 1.16 (Système de grille)** *Un système de grille offre un ensemble de services pour simplifier l'exploitation d'une grille.*

Un système de grille peut offrir des services pour identifier les utilisateurs, gérer les ressources, planifier l'exécution des applications, traiter les défaillances, *etc*. La conception d'un système de grille doit répondre à deux principaux objectifs.

**Simplicité.** Pour rendre l'utilisation d'une grille attractive, il faut que l'utilisateur puisse soumettre une application à la grille aussi simplement que sur son ordinateur personnel. Le système de grille doit donc fournir un ensemble complet de services pour gérer l'application pendant son cycle de vie sur la grille, de la recherche de ressources à l'obtention des résultats. Le critère de transparence est important pour permettre une utilisation simple. Il implique notamment qu'une application doit pouvoir être exécutée sur la grille sans modification préalable.

**Performance.** Une grille de calcul a pour but d'exécuter des applications de calcul haute performance. Les services du système de grille doivent donc viser à optimiser les performances des applications. Par exemple, pour optimiser les performances d'une application parallèle, il est nécessaire d'exécuter tous ses processus sur la même grappe de calcul, ou la même machine massivement parallèle. Ceci doit donc être pris en compte lors de l'allocation de ressources pour ce type d'applications.

L'architecture logicielle sur les nœuds d'une grille de calcul est donc composée de plusieurs couches, comme le montre la figure 1.5. Pour exploiter les ressources d'un nœud de la grille, un système d'exploitation est utilisé. Dans les grilles de calcul et plus généralement dans le domaine du calcul hautes performances [174], le système d'exploitation le plus couramment utilisé est Linux. C'est l'hypothèse que nous prenons dans ce document. Les services du système de grille sont le plus souvent mis en œuvre au-dessus du système d'exploitation, sous la forme d'intergiciels. L'application de grille utilise les services fournis par ces intergiciels pour s'exécuter sur la grille. En fait, l'application utilise ces services pour résoudre certains problèmes liés à l'exécution sur grille, comme par exemple trouver des ressources pour s'exécuter. Cependant les processus de l'application s'exécutent ensuite directement au-dessus du système d'exploitation sur les ressources sélectionnés.

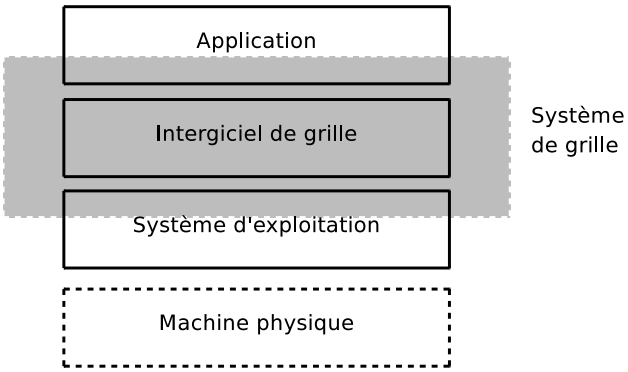



FIG. 1.5 – Les couches logicielles sur un nœud d'une grille de calcul

Sur la figure 1.5, le système de grille s'étend au delà de la couche d'intergiciels. En effet, certains systèmes de grille, comme XtreemOS [50, 129], visent à fournir un système d'exploitation modifié pour être adapté aux besoins des grilles de calcul. De même, l'utilisation d'une interface telle que SAGA [82] (*A Simple API*[1] *for Grid Applications*) permet de développer des applications adaptées aux grilles de calcul et exploitant simplement les fonctionnalités fournies par les intergiciels de grille.

**Cas particulier des grappes de calcul.** Les grappes de calcul offrent un cas particulier d'architecture logicielle dans les grilles. En effet, l'exploitation des ressources d'une grappe de calcul est en général gérée par un gestionnaire de traitement par lots ou par un système à image unique.

Les gestionnaires de traitement par lots, tels que PBS [141], Torque [190], OAR [37], ou encore Sun Grid Engine [66], mettent en œuvre l'ordonnancement des applications à exécuter sur une grappe de calcul à l'aide de files d'attentes. Une machine physique est en général dédiée à l'exécution de ce gestionnaire. Les nœuds sur lesquels sont exécutés les applications sont eux munis d'un système d'exploitation classique. Dans ce cas, le système de grille doit interagir avec le gestionnaire de traitement par lot.

Les systèmes à image unique, tels que OpenSSI [142], OpenMosix [140] ou Kerrighed [44], visent à offrir l'illusion d'une machine unique multiprocesseurs à partir de plusieurs machines physiques. Ainsi, du point de vue de la grille, une grappe de calcul munie d'un système à image unique peut être vue comme un seul nœud ayant une grande quantité de ressources.

**Services d'un système de grille.** L'*Open Grid Forum* [72], dont l'objectif est de développer et promouvoir des standards pour les grilles de calcul, a spécifié une architecture standardisée pour les systèmes de grille : *The Open Grid Service Architecture (OGSA)* [73]. Dans ce document sont décrits les principaux services de haut niveau que peut fournir un système de grille :

– Service de gestion des applications. Ce service a la charge du cycle de vie des applications sur la grille. Ceci comprend la sélection de ressources pour exécuter l'application, le lancement de l'exécution et la supervision des exécutions en cours.

[1] API est l'acronyme pour « *Application Programming Interface* » , c'est-à-dire interface de programmation. Nous utiliserons l'acronyme API dans la suite du document.

– Service de gestion des données. Ce service sert à transférer des données entre ressources de la grille ou à gérer plusieurs copies d'une donnée.
– Service de gestion des ressources. Ce service sert principalement à faire l'interface entre le système de gestion des ressources local, par exemple un gestionnaire de traitement par lots sur une grappe de calcul, et les autres services de grille.
– Service de sécurité. C'est le service en charge de l'authentification des utilisateurs et de la gestion de l'accès aux ressources.
– Service d'information. Ce service doit permettre de diffuser de façon efficace des informations sur les applications, les ressources et les services de la grille.

La mise en œuvre de l'ensemble de ces services dans un système de grille permet de simplifier pour l'utilisateur l'exploitation des ressources de la grille. Ces services ne peuvent être vus comme complètement indépendants les uns des autres car les interactions entre eux sont nécessaires et parfois fréquentes. Par exemple, le service de gestion des applications a besoin du service d'information pour trouver des ressources disponibles pour exécuter une application et du service de gestion des ressources pour exécuter l'application sur les ressources sélectionnées.

Dans OGSA, les services de grilles sont définis comme des services Web (*Web services*) [48] ayant un état. Dans ce document, nous ne considérons pas l'aspect définition de standards associé aux travaux sur OGSA. Nous définissons simplement un service de grille comme un service ayant un état. Un service de grille est une application telle que le décrit la définition 1.3.

### 1.1.6 Un environnement d'exécution hautement dynamique

Une grille de calcul est un environnement d'exécution hautement dynamique en raison de la volatilité des nœuds qui la composent. Nous caractérisons maintenant cette volatilité et mettons en évidence les besoins en tolérance aux fautes.

Plusieurs situations peuvent entraîner des connections et déconnections volontaires. Tout d'abord, le propriétaire de ressources peut à tout moment décider de les retirer de la grille pour en faire un usage exclusif. Ce cas est très fréquent pour des nœuds dont les ressources sont exploitées par la grille en période d'inactivité, comme dans les grilles de calcul pair à pair ou dans les réseaux de stations de travail. De la même manière, une institution peut temporairement retirer ses ressources de la grille pour exécuter des tâches prioritaires. Dans le cadre d'opérations de maintenance, comme des mises à jour matérielles ou logicielles, les nœuds peuvent aussi temporairement être déconnectés de la grille.

La seconde source de volatilité est l'occurrence de défaillances matérielles. Le temps moyen entre deux défaillances ou $MTBF$[2] caractérise la fréquence des défaillances dans un système. Étant donné le nombre de ressource composant une grille de calcul, la fréquence des défaillances y est élevée. En général, lors de l'achat de matériel informatique, celui-ci est garanti trois ans. On peut donc supposer qu'au-delà de trois ans, le risque de subir une défaillance devient grand. Dans le cas d'une grille composée de 25000 nœuds, cela signifie que le $MTBF$ serait environ de un jour.

$$\frac{3\ ans * 365\ jours * 24\ heures}{25000\ nœuds} \approx 1\ jour\ et\ 1\ heure$$

Ce calcul simple n'a pas pour but d'être précis. En effet, il est supposé ici que la probabilité de défaillance est linéaire au cours du temps, ce qui est assez loin de la réalité.

[2] Mean Time Between Failures

Cependant il met clairement en évidence l'importance du problème des défaillances dans les systèmes distribués de grande taille comme les grilles de calcul. Ces défaillances matérielles peuvent affecter un nœud de la grille ou le matériel servant à l'interconnexion des nœuds. Dans le second cas, ce sont tous les nœuds dépendant de ce matériel qui sont déconnectés de la grille.

Le problème lié aux déconnections de nœuds est que des applications ou des services de la grille peuvent être en cours d'exécution sur les nœuds se déconnectant. Comme nous le décrivons en détail dans la partie 1.2, la déconnexion d'un nœud de la grille peut entraîner une perte d'information pour les applications ou services s'exécutant sur ce nœud, et compromettre leurs bon fonctionnement.

Dans le cas de déconnections volontaires, il peut être exigé de demander une autorisation de déconnexion préalablement. Dans ce cas, le système de grille est prévenu que la déconnexion va avoir lieu et peut exécuter les actions nécessaires pour éviter toute perte d'information, avant d'autoriser la déconnexion. Cependant, si la déconnexion est liée à une défaillance, cette solution n'est pas applicable.

Si aucune mesure particulière n'est prise pour traiter cet évènement, les ressources d'une grille de calcul deviennent inexploitables. En effet, en cas de défaillance d'un nœud où s'exécute un ou plusieurs processus d'une application distribuée, la conséquence peut être la défaillance de toute l'application distribuée. L'utilisateur n'obtient alors pas le résultat qu'il attendait. De plus, toutes les ressources consommées par les autres processus de cette application distribuée, le sont inutilement. Si c'est un processus appartenant à un service de grille qui s'exécutait sur le nœud défaillant, ce service devient lui aussi défaillant. Si ce service est indispensable au bon fonctionnement du système de grille, ce sont toutes les ressources de la grille qui deviennent potentiellement inexploitables. C'est pourquoi nous avons besoins de mécanismes de tolérance aux fautes pour être capables de traiter les conséquences d'une déconnexion subie.

## 1.2 La tolérance aux fautes dans les systèmes distribués

Dans cette section, nous présentons la problématique de la tolérance aux fautes dans les systèmes distribués et plus particulièrement dans les grilles de calcul. Dans un premier temps, nous définissons le vocabulaire utilisé dans ce domaine. Puis nous présentons le modèle de faute adapté au contexte des grilles de calcul. Enfin nous dressons un panorama des techniques de tolérance aux fautes.

### 1.2.1 Faute, erreur et défaillance

Avant d'aller plus loin dans la réflexion sur la tolérance aux fautes dans les grilles de calcul, il est important de définir les termes et notions se rapportant à ce domaine. Pour ce faire, nous nous appuyons sur les travaux présentés par Laprie [114].

La défaillance d'un système informatique se définit par rapport au service qu'il fournit. La spécification du service fourni par un système informatique est un accord entre les fournisseurs et les utilisateurs du système sur la description du comportement attendu pour ce système. Un comportement non conforme à cette description est une défaillance du système.

**Définition 1.17 (Défaillance)** *Un système est défaillant quand son comportement ne suit pas sa spécification.*

**Définition 1.18 (Erreur)** *Une erreur est un état interne du système susceptible de provoquer une défaillance.*

**Définition 1.19 (Faute)** *Une faute est une action ou un évènement pouvant entraîner une erreur.*

Il existe deux principales catégories de fautes : les fautes d'origine matérielle, par exemple un court-circuit ou une surchauffe, et les fautes d'origine humaine qui peuvent être liées à une mauvaise conception du système ou à une mauvaise action, effectuée par inadvertance ou volontaire, sur ce système.

Une erreur est la manifestation d'un point vue interne au système d'une faute et une défaillance la manifestation d'un point de vue externe au système d'une erreur. Il existe donc une relation de causalité entre faute, erreur et défaillance.

Un système complexe peut être vu comme un ensemble de sous-systèmes le composant. La défaillance d'un sous-système est alors une faute pour le système auquel il appartient. Les notions de faute et de défaillance dépendent donc du point de vue.

Prenons l'exemple, illustré par la figure 1.6, d'une application parallèle exécutée sur une grille de calcul. Cette application est composée d'un ensemble de processus distribués sur des nœuds de la grille, qui doivent par moment communiquer entre eux pour échanger leurs données. Une faute matérielle se produit sur un nœud exécutant un processus de l'application : un disque dur sur ce nœud est défaillant. Les données présentes sur ce disque ne sont alors plus accessibles. Une tentative d'accès à ces données par le processus va activer l'erreur et entraîner la défaillance du processus. Du point de vue de l'application parallèle, la défaillance d'un processus est une faute créant une erreur latente : on ne peut plus communiquer avec le processus fautif. Les autres processus de l'application peuvent continuer leur exécution. Cependant lors de la prochaine phase de communication, l'erreur va être activée car les processus non fautifs vont tenter de communiquer avec le processus fautif mais ne vont par réussir à récupérer les données de ce processus nécessaires à la poursuite de l'exécution. L'application parallèle ne peut pas terminer son exécution et rendre le résultat attendu à l'utilisateur : elle est défaillante.

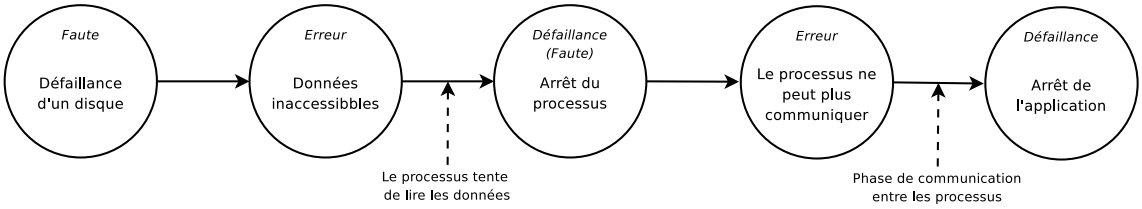



Fig. 1.6 – Illustration des relations de causalité entre faute, erreur et défaillance : exemple d'une application parallèle s'exécutant sur une grille de calcul

### 1.2.2 Les fautes dans les grilles de calcul

**Les modèles de fautes** Plusieurs modèles de fautes sont décrits dans la littérature. On peut en distinguer trois principaux : les fautes par arrêt total, les fautes par omission et les fautes byzantines. Pour spécifier ces modèles, nous considérons un système composé d'un ensemble de composants communiquant par messages.

**Définition 1.20 (Faute par arrêt total)** *Le composant fonctionne conformément à sa spécification jusqu'à la faute par arrêt total [85] où il s'arrête de fonctionner.*

**Définition 1.21 (Faute par omission)** *Lors d'une faute par omission [146], le composant peut* oublier *certains messages en émission ou en réception.*

**Définition 1.22 (Faute byzantine)** *Un composant au comportement byzantin [112] peut envoyer des messages qui ne suivent pas sa spécification. Ceci inclut des comportements malveillants.*

Ces trois types de fautes couvrent l'ensemble du spectre des fautes pouvant intervenir dans un système informatique, une faute par arrêt total étant le cas le plus simple et une faute byzantine étant le cas le plus complexe à traiter. Il est important de choisir le modèle adapté au contexte d'étude car plus le modèle de faute est complexe, plus les mécanismes à employer pour traiter ces fautes sont coûteux.

**Un modèle de faute pour la grille** Comme nous l'avons vu précédemment, étant donné l'échelle des grilles de calcul, les risques de subir une défaillance matérielle sont élevés. La défaillance d'une ressource physique a pour conséquence la défaillance des processus liés à cette ressource, comme illustré par la figure 1.6. Du point du vue de l'application ou du service auquel appartient ce processus, une telle défaillance est une faute par arrêt total.

Dans le cas d'une grappe de calcul, toutes les machines de la grappe sont regroupées dans le même lieu et dépendent généralement du même système de refroidissement. La défaillance du système de refroidissement a alors pour conséquence une surchauffe pour l'ensemble des machines de la grappe, qui doivent donc s'arrêter. On parle alors de défaillances corrélées. Ce cas doit être pris en compte dans le contexte des grilles de calcul.

Les grilles de calcul sont composées de nœuds et de liens d'interconnexion. Une faute par omission dans ce contexte serait liée à la surcharge temporaire de certains liens d'interconnexion. Par exemple, en cas de congestion sur un *switch* celui-ci peut être amené à ignorer certains paquets. Cependant nous considérons que les canaux de communications entre processus de la grille sont équitables.

**Définition 1.23 (Canal de communication équitable)** *Si un canal de communication entre deux processus non fautifs est équitable et si un message est émis une infinité de fois par un processus sur ce canal, un sous ensemble de ces messages est reçu par l'autre processus. De plus, un message doit avoir été émis par un processus pour pouvoir être reçu.*

Cette deuxième propriété signifie que le canal de communication ne peut pas créer de messages.

En mettant en place des mécanismes de réémission de messages, des canaux de communication équitables peuvent être transformés en canaux de communication fiables. Ces mécanismes sont généralement mis en œuvre au niveau des couches basses de la pile logicielle des protocoles de communication.

**Définition 1.24 (Canal de communication fiable)** *Si un canal de communication entre deux processus non fautifs est fiable, un message émis par un processus sur ce canal est reçu par l'autre processus.*

L'utilisation de numéros de séquence permet de mettre en œuvre des canaux de communication FIFO[3] à partir de canaux de communication fiables. Le protocole TCP [1], par exemple, offre ces propriétés.

[3] « First In First Out »

Le matériel servant à l'interconnexion des ressources de la grille peut lui aussi subir des défaillances matérielles. Dans ce cas, il devient impossible de communiquer avec les nœuds dépendant du matériel défaillant. À nouveau, nous considérons qu'il s'agit d'une défaillance corrélée par arrêt total de ces nœuds.

Concevoir un système capable de supporter des fautes byzantines est important principalement si ce système peut faire l'objet d'attaques. Comme nous l'avons vu dans la section 1.1.1, le partage de ressources dans une grille de calcul se fait dans le cadre d'une organisation virtuelle. Celle-ci se matérialise par un contrat entre les membres composant l'organisation virtuelle, définissant les comportements autorisés au sein de cette organisation virtuelle. De plus, l'utilisation d'un service dans la grille est régie par des accords sur la qualité de service (*Service Level Agreement* [116]) passés entre le service et ses utilisateurs, garantissant le comportement fourni par ce service. Enfin, des mécanismes d'authentification et d'autorisation [76] limitent les actions possibles pour les utilisateurs de la grille à un cadre prédéfini. Étant donné cet ensemble de garanties, nous ne considérons pas les fautes byzantines au cours de notre étude.

### 1.2.3 Fondements de la tolérance aux fautes dans les systèmes distribués

Les deux manières de traiter les fautes dans un système sont d'utiliser des techniques permettant de prévenir les fautes ou des techniques permettant de traiter les conséquences des fautes. La prévention des fautes (*fault avoidance*) inclut l'ensemble des techniques permettant de fournir du code fiable, *i.e.* qui ne contient pas d'erreurs. Cependant, il est impossible de prévenir toutes les fautes, et en particuliers les fautes matérielles, dans le cas d'un système de très grande taille comme une grille de calcul. C'est pourquoi des mécanismes de tolérance aux fautes, capables de traiter les conséquences d'une faute sont nécessaires.

Pour pouvoir traiter les conséquences d'une faute, il faut tout d'abord être capable de détecter cette faute, c'est-à-dire être capable de détecter la défaillance à l'origine de la faute. Avant de présenter les principales techniques de tolérance aux fautes existantes, nous faisons une courte introduction à la problématique des détecteurs de défaillance.

**Les détecteurs de défaillance** Détecter les défaillances dans un système distribué asynchrone, *i.e.* un système où aucune hypothèse n'est faite sur le temps de transmission des messages ou le temps de calcul des processeurs, est un problème complexe. En effet, il est dans ce cas à priori impossible de distinguer un processus très lent d'un processus défaillant. Ce problème est la raison de l'impossibilité d'une solution déterministe pour le consensus dans un système distribué asynchrone où un processus peut subir une défaillance [70]. Pour résoudre ce problème, Chandra et Toueg [41] ont proposé la notion de détecteur de défaillance non fiable.

Dans la réalité, il n'existe pas de systèmes distribués complètement asynchrones. La plupart des systèmes distribués se comporte de manière quasi-synchrone, *i.e.* les délais dans le système sont bornés la plupart du temps, même s'ils peuvent avoir un comportement asynchrone de temps en temps. Les détecteurs de défaillance non fiables cherchent à exploiter cette propriété. En effet, un détecteur de défaillance non fiable est un *oracle* distribué qui fournit aux processus une vue approximative des défaillances dans le système au cours de l'exécution : le détecteur de défaillance fournit une liste des processus suspectés d'avoir subi une défaillance. Comme il est impossible de distinguer un processus lent d'un processus défaillant dans un système asynchrone, le détecteur de défaillance va faire des erreurs : il va suspecter des processus non fautifs et ne pas suspecter des processus

fautifs. Les deux propriétés caractérisant les détecteurs de défaillance sont la complétude et la précision. Parmi les huit classes de détecteurs de défaillance définies par Chandra et Toueg, $\diamond S$ est celui ayant les propriétés minimales pour résoudre le problème du consensus : complétude forte et précision ultimement[4] faible.

**Propriété 1.1 (Complétude forte)** *Tous les processus défaillants sont ultimement suspectés par tous les processus non défaillants.*

**Propriété 1.2 (Précision ultimement faible)** *Certains processus non défaillants ne sont ultimement jamais suspectés par les processus non défaillants.*

Si le système se comporte de manière quasi-synchrone pendant suffisamment longtemps, le détecteur de défaillance peut fournir ces propriétés. Des travaux se sont intéressés à la mise en œuvre de détecteurs de défaillance dans les grilles de calcul [19, 51]. La mise en œuvre de détecteurs de défaillance est hors du champ d'étude de ce document.

**Tolérance aux fautes par duplication** La défaillance d'un processus ou d'une machine physique entraîne une perte d'informations, par exemple des données ou l'état du processus. Le seul moyen d'éviter cette perte d'information est de dupliquer les données dans un endroit où elles ne seront pas altérées par la défaillance. Les techniques de tolérance aux fautes dans les systèmes distribués se fondent donc sur des techniques de duplication des informations.

### 1.2.4 Les techniques de tolérance aux fautes

Nous présentons maintenant les deux familles de techniques de tolérance aux fautes existantes, *i.e.* les techniques de recouvrement arrière et les techniques de duplication. Nous décrivons les propriétés de chacune d'entre elles, ainsi que les problèmes techniques qu'elles soulèvent. Puis nous étudions les cas d'utilisation de ces techniques.

#### 1.2.4.1 Techniques de recouvrement arrière

Les techniques de recouvrement arrière se fondent sur la sauvegarde de points de reprise.

**Définition 1.25 (Point de reprise)** *Un point de reprise d'une application est un ensemble de données décrivant l'état de l'application.*

Des points de reprise de l'application sont enregistrés sur un support de stockage stable durant l'exécution de l'application pour pouvoir, en cas de défaillance, redémarrer l'application à partir d'un de ces points de reprise. Ainsi l'application ne redémarre pas depuis zéro après une défaillance. Elle effectue un retour arrière dans l'état sauvegardé dans un point de reprise.

**Définition 1.26 (Retour arrière)** *Un retour arrière correspond au redémarrage de l'application depuis un état antérieur à l'état au moment de la défaillance.*

[4] Il est difficile de traduire l'adverbe anglais *eventually*. La meilleure traduction serait d'utiliser l'expression « *au bout d'un certains temps* » . Dans ce document, nous choisissons de traduire cette notion par l'adverbe *ultimement*

**Définition 1.27 (Stockage stable)** *Un support de stockage stable est une abstraction représentant une solution de stockage de données assurant l'intégrité et la disponibilité des données en dépit des défaillances et fournissant des opérations de lecture et d'écriture atomiques. La manière de mettre en œuvre ce type de support dépend du modèle de faute considéré [105, 113].*

Le cas le plus simple est celui d'une application mono-processus. Dans ce cas sauvegarder un point de reprise de l'application se résume à sauvegarder les informations nécessaires au redémarrage du processus. Ceci peut être fait en collaboration avec l'application si celle-ci a été conçue pour pouvoir le faire. Dans ce cas, c'est l'application qui détermine l'ensemble des informations nécessaires à son redémarrage.

La solution alternative consiste à utiliser un outil extérieur à l'application pour sauvegarder les points de reprise [99, 108]. Ce type de solutions transparentes pour l'application, a l'avantage de simplifier le travail du concepteur de l'application et de fournir une solution de sauvegarde de points de reprise à des applications qui n'ont pas initialement été conçues dans ce but. Le principe est alors de sauvegarder l'image du processus du point de vue du système d'exploitation (mémoire, registres, *etc.*) pour être capable de le redémarrer plus tard. Il n'est dans ce cas pas nécessaire d'avoir connaissance du fonctionnement interne de l'application.

**Applications distribuées** Pour les applications distribuées, la sauvegarde de points de reprise est une tâche plus complexe car les communications entre les processus de l'application créent des dépendances entre eux. Il faut, après une défaillance, être capable de rétablir l'application dans un état global cohérent.

**Définition 1.28 (État global cohérent)** *Un état global cohérent d'une application après une défaillance est un état qui aurait pu être observé durant l'exécution normale de l'application [42].*

Le problème des dépendances entre processus est illustré par la figure 1.7. Dans cet exemple, nous étudions l'état global formé par les points de reprise de trois processus d'une application distribuée. Imaginons que ces trois processus subissent une défaillance et redémarrent depuis leur dernier point de reprise. Dans le cas de la figure 1.7(b), l'état courant du processus $p1$ reflète la réception du message $m0$ alors que l'état du processus $p0$ ne reflète pas son émission. $p1$ est un processus orphelin.

**Définition 1.29 (Processus orphelin)** *Un processus orphelin est un processus dont l'état dépend d'un message qui n'a pas été envoyé. Par extension, nous appelons message orphelin, le message à l'origine de la création du processus orphelin.*

En fait, le processus émetteur du message a fait un retour arrière dans un état précédent l'émission de ce message. Le problème est qu'il est à priori impossible de déterminer si le processus va émettre le même message lors de sa réexécution, des évènements extérieurs comme la réception de nouveaux messages pouvant modifier son comportement. Après une défaillance, il faut donc rétablir l'application dans un état sans processus orphelin.

**Propriété 1.3** *Un état global cohérent est un état sans processus orphelin.*

Sur l'exemple de la figure 1.7(a), l'état global formé par les trois points de reprise est cohérent. Dans cet état, le message $m0$ a été envoyé mais par encore reçu, ce qui est bien sûr un état possible lors d'une exécution sans défaillance.

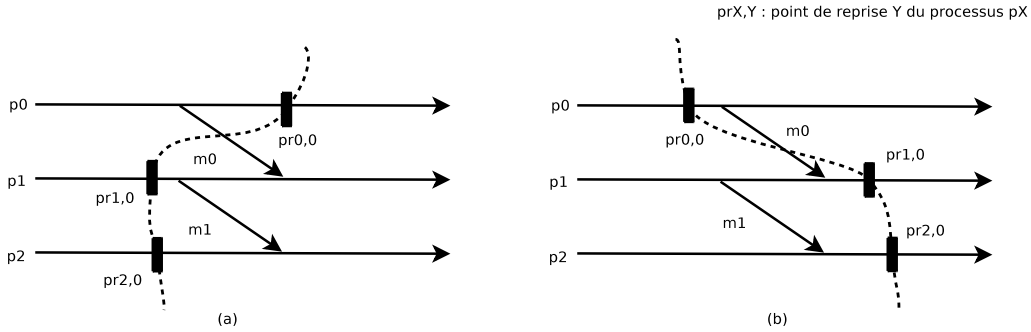



FIG. 1.7 – Exemple d'un état global cohérent et d'un état global non cohérent

Pour sauvegarder un point de reprise d'une application distribuée, il faut donc non seulement sauvegarder l'état des processus mais aussi prendre en compte l'état des canaux de communication. Elnozahy et al. [63] décrivent l'ensemble des techniques de retour arrière existantes pour les applications distribuées communicant par message.

La solution la plus basique est la sauvegarde de points de reprise non-coordonnés [20, 199]. Les points de reprise des processus de l'application sont sauvegardés indépendamment les uns des autres. Chaque processus peut ainsi être sauvegardé au moment le plus approprié, par exemple pour minimiser la taille du point de reprise. Un état global cohérent est calculé seulement au redémarrage après une défaillance. Ce type de solution peut souffrir de l'*effet domino*, comme illustré par la figure 1.8. Après la défaillance du processus $p1$, celui-ci peut à priori redémarrer depuis son dernier point de reprise $pr1,1$. Cependant ce retour arrière rend le message $m6$ orphelin et implique donc le retour arrière du processus $p2$ en $pr2,1$. Ce nouveau retour arrière rend le message $m5$ orphelin et implique donc le retour arrière du processus $p0$ en $pr0,0$. Les retours arrières vont continuer à s'enchaîner ainsi. Sur cet exemple, on peut constater que le seul état global cohérent est le redémarrage de tous les processus depuis zéro. Tous les points de reprise qui ont été sauvegardés au cours de l'exécution de l'application sont donc inutiles. C'est cette invalidation en cascade des points de reprise qui est appelée *effet domino* [152].

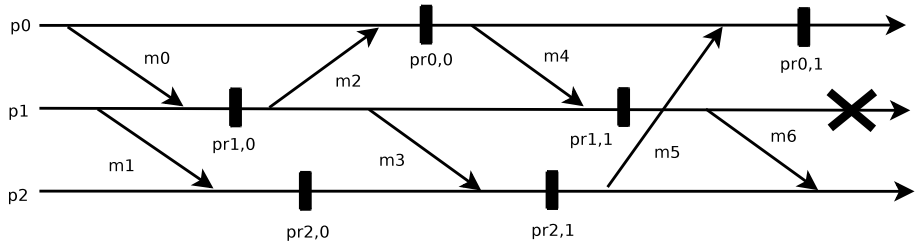



FIG. 1.8 – Effet domino avec des points de reprise non coordonnés

Pour éviter l'effet domino, les techniques de points de reprise coordonnés [42] coordonnent les processus de l'application avant la sauvegarde des points de reprise pour assurer que ceux-ci forment un état global cohérent. Une solution alternative, la sauvegarde de points de reprise induite par les communications [198], consiste à analyser les schémas de communication de chaque processus pour déterminer à quel moment un point de reprise valide, *i.e.* un point de reprise qui pourra faire parti d'un état global cohérent, peut être sauvegardé.

Enfin, la sauvegarde de points de reprise non coordonnée peut être associée avec des techniques d'enregistrement de messages pour éviter l'effet domino. Ces techniques sont classées en trois familles [6], pessimiste, optimiste et causal, offrant des compromis différents entre performance lors d'une exécution sans défaillance et performance au redémarrage après une défaillance.

Nous revenons en détail sur ces différentes techniques de recouvrement arrière dans le chapitre 5, consacré à l'étude de ces techniques dans le contexte des grilles de calcul.

#### Relations avec le monde extérieur

**Définition 1.30 (Monde extérieur)** *Le* monde extérieur *regroupe l'ensemble des entités avec lesquelles une application peut interagir mais qui ne font pas partie de cette application.*

En général, un message envoyé au monde extérieur ne peut pas être invalidé lors d'un retour arrière. Un exemple simple illustrant ce problème est celui d'une imprimante : une fois un caractère écrit par l'imprimante, il est impossible de l'annuler. C'est pourquoi, avant d'envoyer un message vers le monde extérieur, il est nécessaire de s'assurer que celui-ci ne sera jamais invalidé par un retour arrière. Ce problème est appelé le problème de validation des sorties (*output commit*) [184]. Une solution simple pour résoudre ce problème est de sauvegarder un point de reprise coordonné de l'application avant chaque envoi de message au monde extérieur. Cependant, cette solution peut devenir coûteuse quand les interactions avec le monde extérieur sont fréquentes.

### 1.2.4.2 Techniques de duplication

Les techniques de duplication consistent à avoir plusieurs copies identiques d'un objet, appelées duplicatas. Ces techniques sont en général appliquées à un processus. Ainsi si un processus subit une défaillance, un duplicata de ce processus peut le remplacer. Nous présentons les deux principales techniques de duplication existantes : la duplication passive et la duplication active. Comme nous le verrons dans la section 1.2.4.3, ce type de solution est généralement appliqué à des services. C'est pourquoi nous prenons l'exemple d'un service en interaction avec des clients pour illustrer notre description.

**Duplication passive** Avec une solution de type duplication passive (*Primary-Backup*) [36], un duplicata du service joue un rôle particulier. Il est appelé duplicata primaire. C'est le seul duplicata qui interagit avec les clients comme le montre la figure 1.9. Les autres duplicatas n'interagissent qu'avec le duplicata primaire et sont appelés duplicatas de sauvegarde. Après avoir traité la requête du client (1), le duplicata primaire doit s'assurer que les duplicatas de sauvegarde sont à jour avant d'envoyer la réponse au client. Pour cela, il envoie un message contenant les informations de mise à jour relatives au traitement de la requête aux duplicatas de sauvegarde (2). Quand il sait que les duplicatas ont été mis à jour (3), il envoie la réponse au client (4).

En cas de défaillance du duplicata primaire, il est remplacé par un des duplicatas de sauvegarde qui devient le nouveau duplicata primaire. Le client doit alors être averti de ce changement de duplicata primaire pour pouvoir s'y connecter et réémettre les requêtes pour lesquelles il n'a pas obtenu de réponse.

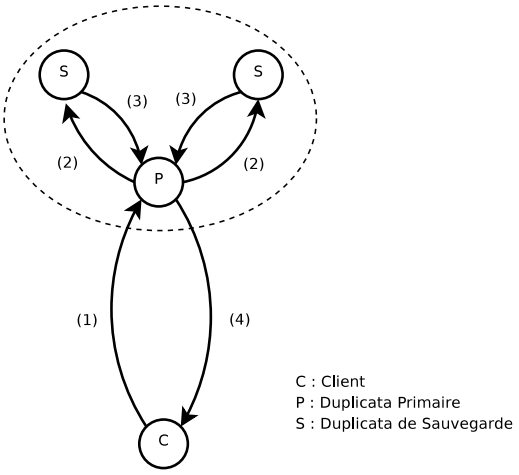



FIG. 1.9 – Duplication passive d'un service

**Duplication active** Avec une solution de type duplication active, aussi appelée approche de type machine à état [168], tous les duplicatas du service jouent le même rôle comme l'illustre la figure 1.10. Cette approche requiert un comportement déterministe du service. Ainsi, pour assurer la cohérence entre les duplicatas, il suffit d'assurer que ceux-ci délivrent le même ensemble de messages dans le même ordre. La requête du client est donc diffusée à l'ensemble des duplicatas du service (1). Les duplicatas se mettent ensuite d'accord sur l'ordre dans lequel délivrer ces requêtes (2). Chaque duplicata traite ensuite la requête et envoie sa réponse au client (3). Le client peut alors utiliser la première réponse qui lui parvient et ignorer les suivantes.

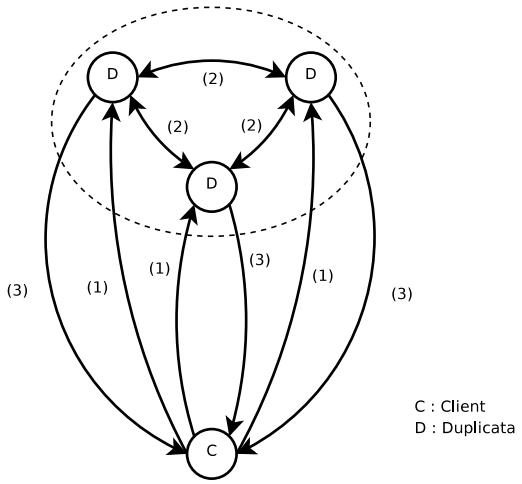



FIG. 1.10 – Duplication active d'un service

Contrairement aux techniques de duplication passive, les techniques de duplication active rendent la défaillance d'un duplicata du service complètement transparente pour le client, puisque d'autres duplicatas actifs sont toujours disponibles pour répondre aux requêtes. Cependant, les techniques de duplication active ne peuvent être utilisées que pour des services ayant un comportement déterministe. Les solutions de duplication passive n'ont pas cette limitation. De plus, comme on peut le constater en comparant les figures

1.9 et 1.10, la duplication active implique des communications plus nombreuses.

#### 1.2.4.3 Niveau de disponibilité

**Définition 1.31 (Haute disponibilité)** *Un système ayant la capacité de satisfaire sa spécification et capable de masquer certaines fautes prédéfinies est hautement disponible [156].*

La haute disponibilité regroupe l'ensemble des solutions comprises entre la disponibilité de base, *i.e.* un système satisfaisant sa spécification mais ne masquant pas les fautes, et la disponibilité continue, *i.e.* un système capable de masquer les fautes mais aussi les arrêts de machines prévus.

Les solutions de tolérance aux fautes présentées précédemment offrent différents niveaux de haute disponibilité, c'est-à-dire qu'elles masquent plus ou moins les fautes.

- Les techniques de recouvrement arrière fondées sur la sauvegarde de points de reprise impliquent en général un retour arrière des processus après une défaillance. La défaillance n'est donc que partiellement masquée. De plus les processus défaillants doivent être réinitialisés à partir des informations contenues dans leurs points de reprise au redémarrage. Si des processus ont de fortes dépendances entre eux, comme dans le cas d'une application distribuée, il est alors préférable d'utiliser un protocole entre ces processus pour éviter l'effet domino.
- La duplication passive offre un meilleur masquage des fautes. En effet, un processus dupliqué passivement ne fera pas de retour arrière en cas de faute, puisqu'un duplicata de sauvegarde ayant un état cohérent avec le processus actif va pouvoir le remplacer. Il n'est donc pas nécessaire dans ce cas de coordonner les processus ayant des dépendances entre eux. Cependant le duplicata de sauvegarde n'est que partiellement initialisé. Il faut donc un certains temps avant que le système soit rétabli dans le même état qu'avant la faute. Il faut notamment que les processus en interaction avec le processus fautif se connectent explicitement avec le nouveau processus.
- La duplication active rend les défaillances de processus complètement transparentes pour l'extérieur. Une faute n'entraîne pas de retour arrière et ne nécessite pas de reconnexion de la part des autres processus.

Le coût de mise en œuvre de ces solutions augmente avec le niveau de disponibilité fourni. En effet, alors que les techniques de recouvrement arrière nécessitent seulement la sauvegarde d'informations sur support stable, les techniques de duplication nécessitent l'exécution de processus en parallèle sur plusieurs machines physiques. Ainsi, avoir trois duplicatas d'un processus, implique l'utilisation du triple de ressources par rapport à une exécution simple du processus.

C'est pourquoi les techniques de duplication sont en général une solution trop coûteuse pour fournir de la tolérance aux fautes à des applications distribuées. Il est préférable d'utiliser des techniques de recouvrement arrière. Les techniques de duplication ne sont utilisées que pour des services ayant de nombreuses interactions avec le monde extérieur et ne pouvant donc supporter de retour arrière. Il est donc important de bien identifier les besoins en tolérance aux fautes d'un composant pour lui appliquer la solution la plus appropriée.

## 1.3 Synthèse

Une grille de calcul est un système distribué regroupant un grand nombre de ressources hétérogènes appartenant à différents domaines d'administration et partagées au sein d'or-

ganisations virtuelles. Plusieurs architectures de grilles existent. Pour notre étude, nous considérons l'architecture la plus générique : une grille hétérogène. Ce type d'architecture permet de répondre aux besoins en ressource de calcul des applications de calcul scientifique. Cependant exploiter les ressources d'une grille de calcul n'est pas simple car le nombre d'utilisateurs de la grille et donc d'applications s'exécutant sur celle-ci peut être très grand. De plus, il faut être capable de prendre en compte l'hétérogénéité et la volatilité des nœuds de la grille, conséquences de sa taille. C'est pourquoi un système de grille est utilisé. Son rôle est de simplifier l'exploitation de la grille par les utilisateurs tout en optimisant les performances.

Nous nous intéressons aux problèmes liés à la volatilité des nœuds de la grille. Cette volatilité peut être due à des connections et des déconnections volontaires ou à des défaillances. Le cas problématique est celui des défaillances. Étant donné la taille d'une grille de calcul, les risques de défaillances y sont élevés. La défaillance d'un nœud de la grille peut entraîner la défaillance d'applications et/ou de services du système de grille.

Comme les applications de calcul scientifique peuvent avoir un temps d'exécution très grand et être distribuées sur un grand nombre de ressources, la probabilité qu'elles subissent une défaillance au cours de leur exécution sur la grille est très élevée, compromettant leur bonne terminaison. De même, la défaillance d'un service du système de grille peut empêcher les applications de fonctionner correctement, dans le cas d'un service de gestion de données, voir même empêcher les utilisateurs de la grille de soumettre son application, dans le cas d'un service de gestion des applications. Traiter les conséquences des défaillances de nœuds est donc indispensable pour qu'une grille ne soit pas qu'un grand ensemble de ressources inexploitables. C'est pourquoi ce document s'intéresse aux solutions permettant d'exécuter de manière fiable des applications dans les grilles de calcul.

Les techniques de tolérance aux fautes ont pour objectif de traiter les conséquences d'une défaillance. Les trois familles de technique de tolérance aux fautes existantes sont les techniques de retour arrière, la duplication passive et la duplication active. Chacune de ses techniques offre des niveaux de disponibilité différents. Plus le niveau de disponibilité offert est élevé, plus le coût de mise œuvre de la solution, en terme de consommation de ressources, est élevé. Cependant, plus la quantité de ressources nécessaires à la tolérance aux fautes est élevée, moins celle disponible pour l'exécution des applications est grande. Dans le domaine du calcul haute performance, c'est donc un enjeu important. C'est pourquoi il est indispensable de spécifier correctement les besoins en tolérance aux fautes pour être capable de fournir la solution la plus appropriée.

Travailler sur des techniques de tolérance aux fautes nécessite préalablement de définir le modèle de faute considéré. Nous considérons que dans une grille les garanties en terme de sécurité offertes par les organisations virtuelles permettent d'écarter les fautes byzantines. Nous choisissons donc un modèle de faute par arrêt total. Ces fautes, conséquences de défaillances matérielles, peuvent être indépendantes, comme dans le cas de la défaillance d'un disque sur une machine physique, ou corrélées, comme dans le cas d'une défaillance d'un système de refroidissement affectant tous les nœuds d'une grappe de calcul. Nous considérons des canaux de communications fiables et FIFO. La défaillance d'un lien d'interconnexion est vue comme la défaillance des nœuds dépendant de ce lien. Enfin nous supposons disposer de mécanismes de détection de défaillances répondant au moins à la spécification du détecteur de défaillance non fiable $\diamond S$.

Nous voulons fournir des solutions pour assurer une utilisation simple et optimisée des ressources de la grille en dépit de l'occurrence de défaillances. Cette optimisation passe par la mise en œuvre de solutions adaptées au contexte des grilles de calcul, c'est-à-dire prenant en compte la taille de la grille et son hétérogénéité. La simplicité des solutions

proposées doit être jugée du point de vue des utilisateurs. C'est une condition indispensable pour rendre les grilles de calcul attractives. En dépit de l'occurrence de défaillances sur la grille, nous voulons être capable d'assurer aux utilisateurs la bonne terminaison de leur application et l'obtention des résultats associés et ce, le plus rapidement possible.

Si les travaux présentés dans ce document sont menés dans le contexte des grilles de calcul, la problématique de la gestion de la volatilité des ressources dans un environnement distribué de grande taille et hétérogène peut être considérée comme une problématique commune à d'autres types de systèmes distribués. Aussi, les solutions proposées dans ce document peuvent être applicables à d'autres systèmes distribués dont le modèle de fautes est le même que celui utilisé ici.

# Chapitre 2

# Exécution fiable d'applications sur grille de calcul

Nous avons montré dans le chapitre 1 que la gestion des défaillances dans les grilles de calcul est un enjeu majeur. En effet, si les conséquences de ces défaillances ne sont pas correctement traitées, la bonne terminaison des applications des utilisateurs de la grille se trouve compromise. Les ressources fournies par une grille de calcul sont alors inexploitables. Nous présentons dans ce chapitre l'approche que nous proposons pour répondre à cette problématique.

L'organisation de ce chapitre est la suivante. Dans le paragraphe 2.1, nous explicitons les besoins en tolérance aux fautes dans les grilles de calcul. Le paragraphe 2.2 présente les contraintes liées à la nature des grilles de calcul à prendre en considération dans notre réflexion. Nous présentons notre approche dans le paragraphe 2.3. Enfin le paragraphe 2.4 présente une synthèse des idées développées dans ce chapitre.

## 2.1 Définition des besoins

Dans ce paragraphe, nous définissons les besoins en tolérance aux fautes du point de vue des utilisateurs de la grille.

### 2.1.1 Fiabilité

L'utilisateur soumettant une application sur la grille veut obtenir le résultat de cette application. Il faut donc être capable d'assurer en dépit des défaillances dans l'infrastructure de grille que son application se termine bien. Ces défaillances peuvent affecter directement les processus de l'application ou affecter un service de grille utilisé par l'application. Ces deux cas doivent être traités.

De plus, le délai d'obtention des résultats peut être soumis à des contraintes. L'utilisateur peut avoir besoin de ses résultats avant une date limite. Il faut donc que le temps d'exécution des applications dans la grille soit proche du temps d'exécution supposé dans un environnement sans défaillance. La solution simple consistant à redémarrer l'application depuis zéro à chaque fois qu'elle subit une défaillance doit donc être écartée.

### 2.1.2 Simplicité d'utilisation

Il n'est pas réaliste d'exiger des utilisateurs qu'ils surveillent et prennent en charge les conséquences des défaillances que subissent leurs applications qui peuvent être distribuées

sur un grand nombre de ressources dans la grille. En effet, la tolérance aux fautes dans les systèmes distribués est une problématique complexe, pas à la portée de tous les utilisateurs et/ou programmeurs. C'est pourquoi il est souhaitable de leur fournir des solutions traitant les problèmes de tolérance aux fautes dans la grille avec la plus grande transparence possible.

Ce besoin de transparence implique tout d'abord que l'utilisateur ne devrait pas avoir à modifier son application pour assurer son exécution fiable sur la grille. Il implique ensuite que les services de grille doivent être disponibles et simplement accessibles au moment où l'utilisateur en a besoin, et tout au long de l'exécution des applications. Étant donné la taille et la dynamicité d'une grille de calcul, ce deuxième point met en évidence le besoin d'auto-réparation, c'est-à-dire que les conséquences des défaillances doivent pouvoir être traitées de manière automatique, sans intervention humaine.

### 2.1.3 Coût

Comme nous l'avons vu dans le chapitre 1, la tolérance aux fautes se fonde sur de la duplication, et est donc consommatrice de ressources. Cependant les ressources agrégées dans une grille de calcul doivent en priorité servir à l'exécution des applications des utilisateurs. Il faut donc tenter de réduire au maximum les ressources consommées par les mécanismes de tolérance aux fautes.

Plus le niveau de disponibilité offert par une solution de tolérance aux fautes est grand, plus elle est coûteuse en terme de ressources consommées. Le problème à résoudre est donc de trouver le bon compromis permettant d'offrir de bonnes performances aux utilisateurs tout en répondant aux besoins de fiabilité et de simplicité d'utilisation.

## 2.2 Analyse des contraintes liées aux grilles de calcul

Nous présentons maintenant les contraintes liées aux caractéristiques des grilles de calcul que nous devons prendre en compte.

### 2.2.1 Volatilité

La volatilité des nœuds de la grille est causée par les connexions et déconnexions volontaires, et par les défaillances affectant l'infrastructure de grille. Les connexions de nouveaux nœuds ne mettent pas en danger la terminaison des applications s'exécutant sur la grille. Cependant, si aucun mécanisme n'est prévu pour traiter le cas de la déconnexion volontaire d'un nœud, celle-ci doit alors être considérée comme la défaillance du nœud. En effet, le résultat est le même puisque dans les deux cas, les processus s'exécutant sur ce nœud sont perdus.

Pour optimiser les performances des applications sur la grille, l'allocation de ressources prend généralement en compte l'hétérogénéité de la grille notamment pour ce qui concerne les liens d'interconnexion. Pour optimiser les performances d'une application parallèle par exemple, il est préférable d'avoir une latence faible et une bande passante élevée entre les nœuds sur lesquels elle s'exécute. Ceci peut par exemple amener à sélectionner plusieurs nœuds d'une même grappe de calcul pour exécuter une application donnée. Dans ce cas, l'application peut subir la défaillance corrélée de plusieurs nœuds. Il faut donc que les mécanismes de tolérance aux fautes soient capables de traiter ce cas.

### 2.2.2 Passage à l'échelle

La taille d'une grille de calcul peut aller jusqu'à plusieurs dizaines de milliers de ressources. Elle permet d'exécuter des applications distribuées composées de milliers de processus. Les solutions de tolérance aux fautes que nous proposons doivent donc passer à l'échelle. Cela signifie qu'elles doivent offrir de bonnes performances même pour des applications de grande taille.

En outre, plus le nombre de nœuds sur lesquels s'exécute les processus d'une application est élevé, plus la probabilité de défaillances est grande. Les solutions de tolérance aux fautes que nous proposons doivent donc être adaptées à des fréquences de défaillances élevées.

### 2.2.3 Hétérogénéité

Dans le chapitre 1, nous avons vu qu'une grille de calcul peut être composée de ressources hétérogènes comme des ordinateurs personnels et des grappes de calcul. Cette hétérogénéité peut aussi se retrouver dans le logiciel. Il est par exemple fort probable que des ressources appartenant à des domaines d'administration différents ne soient pas équipées des mêmes logiciels. Cependant, des applications peuvent être distribuées sur ces ressources hétérogènes. Il faut alors être capable d'assurer l'exécution fiable de ces applications en prenant en compte cette hétérogénéité.

Cette hétérogénéité se retrouve aussi dans la variété des applications qui peuvent être exécutées sur une grille de calcul. Ces applications peuvent avoir des caractéristiques et des besoins en tolérance aux fautes différents. C'est pourquoi il est souhaitable de proposer des solutions de tolérance aux fautes génériques, capables de s'adapter à la variété des besoins.

## 2.3 Approche proposée

L'objectif principal auquel tente de répondre cette thèse est d'assurer l'exécution fiable sur grille de calcul d'applications de calcul haute performance. Pour atteindre cet objectif, nous proposons un ensemble de solutions qui peuvent être regroupées autour de trois axes principaux :

- Un service de recouvrement arrière pour assurer le redémarrage automatique des applications défaillantes.
- Un cadre pour mettre en œuvre des services de grille hautement disponibles et auto-réparants.
- Un protocole de recouvrement arrière passant à l'échelle pour les applications à échange de messages.

Nous décrivons dans la suite de ce paragraphe, les contributions associées à chacun de ces axes.

### 2.3.1 Un service de recouvrement arrière pour applications distribuées

Pour assurer que les applications exécutées sur une grille se terminent correctement, il faut être capable de redémarrer celles dont l'exécution est interrompue par une défaillance dans l'infrastructure sous-jacente. Telle est la mission de notre service de recouvrement arrière, nommé XtreemGCP [127] et conçu dans le cadre du projet européen XtreemOS [204].

Pour simplifier l'utilisation de la grille, l'utilisateur doit pouvoir être déchargé complètement de la gestion des défaillances. Il est admis que les techniques de recou-

vrement arrière sont les plus adaptées pour la tolérance aux fautes d'applications de calcul haute performance. Un service de recouvrement arrière pour grille de calcul doit pouvoir redémarrer automatiquement les applications défaillantes. Il doit de plus être capable de s'adapter à la variété des caractéristiques des applications et à l'hétérogénéité des ressources sur lesquelles elles s'exécutent. Enfin, dans la perspective de simplifier l'utilisation de la grille, il est souhaitable qu'il puisse fournir des mécanismes de tolérance aux fautes transparents pour les applications qui n'ont pas été conçues pour tolérer les défaillances.

Pour assurer le redémarrage automatique des applications, XtreemGCP prend en charge les interactions nécessaires avec les services du système de grille. Dans le cadre du système XtreemOS, ceci inclut des interactions avec le service de gestion des travaux [47] pour obtenir les ressources nécessaires pour remplacer les ressources défaillantes, ou encore les interactions avec le système de fichier distribué XtreemFS [91] pour le stockage des données générées par la sauvegarde des points de reprise.

Fondé sur une architecture complètement distribuée lui permettant de passer à l'échelle, XtreemGCP est conçu de manière générique pour pouvoir y intégrer dans le futur différents travaux de la communauté en tolérance aux fautes. Cette généricité permet de prendre en charge l'hétérogénéité de la grille et des applications qui s'y exécutent. Ainsi XtreemGCP peut exploiter des mécanismes de tolérance aux fautes qui seraient directement inclus dans l'application ou dans des bibliothèques tolérantes aux fautes utilisées par l'application, comme certaines bibliothèques de communication MPI [30, 166, 93]. De plus, nous proposons une interface générique aux mécanismes de sauvegarde de points de reprise de processus, tels que BLCR [99] ou OpenVZ [143]. Cette interface permet à XtreemGCP d'intégrer simplement plusieurs de ces mécanismes. Ainsi il est capable de gérer des applications distribuées sur des ressources hétérogènes fournissant des mécanismes de sauvegarde de points de reprise de processus différents.

Enfin, pour permettre à l'utilisateur de la grille, qui n'est pas nécessairement informaticien, de rendre sont application tolérante aux fautes sans modifier celle-ci, nous proposons de fournir des solutions de recouvrement arrière appliquées de manière transparente pour l'application. Ainsi XtreemGCP permet de mettre en œuvre différents protocoles de recouvrement arrière pour applications à échange de messages. Pour cela, il se fonde sur des solutions de sauvegarde de points de reprise de processus mises en œuvre par le système d'exploitation des nœuds de la grille [99, 143].

Le chapitre 3 présente la conception du service XtreemGCP. Ces travaux sont le fruit de collaborations avec les membres du projet XtreemOS, et plus particulièrement avec John Mehnert-Sphan et Mickael Schöttner de *Heinrich-Heine University of Düsseldorf*. La mise en œuvre de ce service [127], l'intégration de différents outils de sauvegarde de processus, et la mise en œuvre de protocoles de recouvrement arrière au sein du service [68], ont été réalisées par les partenaires du projet.

### 2.3.2 Un cadre pour la mise en œuvre de services hautement disponibles

La haute disponibilité des services du système de grille est un enjeu majeur. Si un utilisateur doit soumettre une application pour en obtenir les résultats au plus vite, il faut que le service de soumission d'application lui permette de le faire. De même pour qu'un service tel que XtreemGCP soit capable de redémarrer une application défaillante, il faut qu'il soit disponible au moment où l'application subit la défaillance. Cependant ces services s'exécutant eux aussi sur des nœuds de la grille, ils peuvent subir des défaillances. La volatilité des nœuds de la grille peut compromettre la disponibilité de ces services.

Les réseaux pair-à-pair structurés [163, 182] sont une solution attrayante pour la conception de systèmes de grilles adaptées aux grilles de grande taille et dynamiques [103, 155] car ils fournissent des primitives de routage tolérantes aux fautes et passant à l'échelle.

Notre objectif est d'assurer la haute disponibilité des services de grille dans ce contexte. Nous voulons être capable d'assurer la haute disponibilité des services sans intervention humaine pour gérer les conséquences des défaillances. De plus, nous voulons que les services existants puissent être rendus hautement disponibles avec un effort minimal de la part du programmeur. Enfin, dans un souci de simplicité d'utilisation, nous voulons que les défaillances que subissent ces services soient rendues complètement transparentes pour leurs utilisateurs.

Pour rendre les services de grille hautement disponibles, nous optons pour une technique de duplication active car un service en interaction avec le monde extérieur ne peut supporter de retour arrière. De plus, ce choix permet de dupliquer des services à état au comportement déterministe sans modification de ceux-ci. L'étude des services composant le système de grille Vigne [159] nous a montré qu'ils étaient déterministes. Nos travaux portent donc sur la duplication active de service dans un réseau pair-à-pair structuré.

En distribuant les services dupliqués dans une table de hachage distribuée mise en œuvre au dessus du réseau pair-à-pair structuré, il est possible de gérer un grand nombre de service et de répartir la charge des services dupliqués sur l'ensemble des nœuds de la grille. En exploitant les solutions de routage fondées sur des clés, évitant d'associer les services de la grille à des adresses physiques, la duplication active des services et les reconfigurations nécessaires pour traiter les conséquences des défaillances, peuvent être rendues complètement transparentes pour les utilisateurs.

Semias est le cadre que nous avons conçu et mis en œuvre pour rendre des services de grille hautement disponibles et auto-réparants. L'auto-réparation est la capacité d'un système à maintenir son niveau de disponibilité. Elle implique des reconfigurations automatiques pour remplacer les duplicatas défaillants par de nouveaux duplicatas et ainsi garantir la haute disponibilité des services sans intervention humaine.

Dans le chapitre 4, nous présentons l'architecture de Semias. Cette architecture est conçue pour gérer efficacement la volatilité des nœuds de la grille. Nous décrivons notre solution d'auto-réparation pour assurer la haute disponibilité des services dans un environnement très dynamique tout en limitant le coût des reconfigurations. Nous présentons enfin une évaluation de notre prototype sur Grid'5000. Dans le cadre de ces évaluations, nous avons utilisé Semias pour rendre le service de gestion d'application de Vigne hautement disponible et auto-réparant.

Trois stagiaires que j'ai encadré, ont participé aux travaux sur Semias : Rajib Nath, étudiant en Master à *University of Tennessee* (États-Unis), Sébastien Gillot, étudiant en Master à l'Université de Rennes 1, et Stefania Costache, étudiante en Master à *Politehnica University of Bucarest* (Roumanie).

### 2.3.3 Un protocole de recouvrement arrière pour applications à échange de messages de grande taille

Les grilles de calcul peuvent fournir de grandes capacités de ressources de calcul et donc permettre aux utilisateurs d'exécuter des applications de très grande taille. Nombre de ces applications sont fondées sur l'échange de messages. De nombreux protocoles de recouvrement arrière pour applications à échange de messages existent, mais très peu abordent le problème du passage à l'échelle. La sauvegarde de points de reprise coordonnés, technique la plus couramment employée, nécessite le retour arrière de tous les processus

de l'application après la défaillance d'un d'entre eux. Elle n'est pas adaptée pour des applications de grande taille [117] exécutées dans un environnement dynamique.

Nous proposons un protocole à enregistrement de messages optimiste adapté aux applications à échange de messages de grande taille. Les protocoles à enregistrement de messages peuvent par nature passer à l'échelle car ils ne nécessitent aucune coordination entre les processus de l'application en fonctionnement normal. De plus, ils sont adaptés aux environnements dynamiques car ils ne nécessitent pas le redémarrage de tous les processus suite à une défaillance. Enfin Les protocoles à enregistrement de messages optimiste peuvent offrir de bonnes performances car ils permettent d'effectuer les sauvegardes d'information sur support stable en parallèle avec l'exécution des processus de l'application.

Le seul surcoût en terme de performance engendré par les protocoles à enregistrement de messages optimistes est lié à la gestion des informations de dépendance entre les processus de l'application qui sont nécessaires lors du recouvrement pour détecter les processus orphelins. Les protocoles optimistes attachent des informations de dépendance sur les messages de l'application. Pour les protocoles existants capables tolérants plusieurs fautes, la taille de ces informations est proportionnelle à la taille de l'application.

Pour un meilleur passage à l'échelle, nous proposons un nouveau protocole fondé sur l'enregistrement de messages optimiste *actif* qui permet de limiter la taille des informations attachées sur chaque message de l'application tout en conservant la capacité de tolérer plusieurs fautes simultanées.

La gestion centralisée de la sauvegarde des informations de dépendances considérée dans les travaux sur les protocoles à enregistrement de messages est une autre limite au passage à l'échelle. C'est pourquoi nous proposons une solution distribuée pour la sauvegarde des informations de dépendance fondée sur l'utilisation de la mémoire volatile des nœuds sur lesquels est exécutée l'application.

Dans le chapitre 5, nous décrivons le protocole O2P [162] et prouvons qu'il est capable de tolérer plusieurs fautes simultanées de processus de l'application. Nous décrivons ensuite une mise en œuvre de ce protocole et de la solution distribuée que nous proposons pour la sauvegarde des informations de dépendance dans la bibliothèque Open MPI. Enfin nous présentons une évaluation de ces solutions sur la plate-forme Grid'5000.

## 2.4 Synthèse

La volatilité des nœuds et en particulier les risques de défaillance, peuvent compromettre la terminaison des applications s'exécutant sur une grille de calcul, empêchant les utilisateurs d'obtenir les résultats attendus. Il est donc nécessaire de mettre en place des solutions de tolérance aux fautes. L'objectif vers lequel nous devons tendre est de rendre l'exécution d'applications sur grille de calcul aussi simple que si aucune défaillance ne s'y produisait.

Pour cela, nous proposons un service de recouvrement arrière pour la grille capable de redémarrer automatiquement les applications défaillantes sans intervention de l'utilisateur. Ce service est conçu de manière générique pour s'adapter à la variété des applications pouvant être soumises à la grille et à l'hétérogénéité des ressources.

Pour assurer le bon fonctionnement de ce service, et plus généralement pour que les services de grilles soient disponibles pour les utilisateurs et les applications en dépit des défaillances, nous proposons un cadre offrant les propriétés de haute disponibilité et d'autoréparation aux services à états du système de grille. Ce cadre, ne nécessitant que peu de modifications des services existants, assure une gestion transparente des reconfigurations pour les utilisateurs.

Enfin, pour assurer une exécution fiable sur grille de calcul d'applications à échanges de messages de grande taille avec un impact minimal sur les performances de ces applications, nous proposons un protocole à enregistrement de messages optimiste actif. Nous proposons de plus une mise en œuvre distribuée de l'enregistrement des messages pour un meilleur passage à l'échelle.

Ces trois contributions sont décrites dans les chapitres 3, 4, et 5.

# Chapitre 3

# Un service de recouvrement arrière pour la grille

Ce chapitre présente XtreemGCP [127], un service de recouvrement arrière pour grille de calcul. XtreemGCP fournit des solutions de tolérance aux fautes par recouvrement arrière aux applications s'exécutant sur la grille pour assurer leur terminaison en dépit des défaillances.

XtreemGCP est conçu pour pouvoir mettre en œuvre différents algorithmes de sauvegarde de points de reprise pour applications distribuées et redémarrer automatiquement les applications défaillantes. Les solutions de tolérance aux fautes proposées sont transparentes pour les applications à exécuter sur la grille. De plus, pour traiter les problèmes d'hétérogénéité logicielle sur la grille, nous proposons l'utilisation d'une interface générique permettant à XtreemGCP d'exploiter différents outils de sauvegarde de points de reprise de processus présent sur la grille.

Nos travaux sur XtreemGCP ont été menés dans le cadre du projet européen XtreemOS. XtreemGCP s'appuie sur d'autres services de XtreemOS, comme XtreemFS [91] pour le stockage persistant des points de reprise ou le service de gestion des travaux [47] pour le redémarrage automatique des applications défaillantes.

Les travaux présentés dans ce chapitre sont le fruit de collaborations avec les membres du projet XtreemOS, et plus particulièrement avec John Mehnert-Sphan et Mickael Schöttner de *Heinrich-Heine University of Düsseldorf*. La mise en œuvre du service XtreemGCP, effectuée dans le projet XtreemOS, sort du cadre de nos travaux. Pour plus de détails, nous encourageons le lecteur à se reporter vers les travaux du projet XtreemOS.

Ce chapitre se structure de la façon suivante. Dans le paragraphe 3.1, nous détaillons le contexte de ses travaux et ses objectifs. Puis nous mettons en évidence à travers l'étude de l'état de l'art, le besoin de nouvelles solutions pour mettre en œuvre des techniques de recouvrement arrière dans la grille. Le paragraphe 3.2 décrit l'architecture et les principes de fonctionnement de XtreemGCP. Enfin nous dressons un bilan de ces travaux dans le paragraphe 3.3.

## 3.1 Problématique

Pour présenter les défis liés à la mise en œuvre de solutions de recouvrement arrière dans une grille de calcul, nous commençons par présenter le modèle d'application considéré. Puis nous présentons le système de grille XtreemOS [204, 50, 129] qui est le contexte de cette étude. Nous détaillons ensuite les cas d'utilisation d'un service de recouvrement

arrière dans la grille et précisons les objectifs auxquels nous estimons qu'un service de recouvrement arrière pour la grille doit répondre. Nous étudions les travaux apparentés à l'aune de ces objectifs et nous concluons sur la nécessité de la spécification d'un nouveau service de tolérance aux fautes pour la grille capable de répondre à nos besoins.

### 3.1.1 Modèle d'application

Pour ces travaux, nous visons des applications distribuées s'exécutant sur une grille de calcul. Nous ne faisons aucune supposition sur les paradigmes de programmation employés au sein de ces applications. C'est pourquoi nous considérons un modèle simple d'application distribuée où les processus de l'application communiquent en s'échangeant des messages.

Nous avons vu dans le paragraphe 1.1.2 que plusieurs modèles d'applications distribuées existent. Il est maintenant nécessaire de clarifier ce que notre service de recouvrement arrière est capable de faire. Pour cela, nous reprenons les définitions proposées par l'Open Grid Forum [72] (OGF). L'OGF introduit la notion de travail (*job*). Un travail est décrit par un fichier JSDL [12] (*Job Submission Description Language*) qui définit le travail et les besoins qui lui sont associés en terme de ressources. Un seul fichier exécutable est associé à un travail. Nous voyons donc un travail comme l'exécution sur la grille d'une application non distribuée ou d'une application distribuée fortement couplée comme les applications parallèles.

Pour construire des applications de grille plus complexes, composées de plusieurs travaux, l'API SAGA [82] peut être utilisée. SAGA permet de décrire des applications complexes en manipulant des travaux. Des moteurs de *workflow*, comme Salome [165], peuvent aussi être utilisés pour décrire des applications complexes composées de plusieurs travaux et pour décrire précisément des relations de dépendance entre ces travaux.

Notre service de recouvrement arrière vise la sauvegarde de points de reprise de travaux. Nous nous intéressons donc principalement aux applications parallèles. Pour des applications plus complexes comme les applications de couplage de code ou des *workflows*, composées de plusieurs travaux, nous estimons qu'assurer que chacune des parties composant l'application est tolérante aux fautes est suffisant pour rendre l'ensemble tolérant aux fautes. Le problème à résoudre est alors celui de la validation des sorties décrit dans le paragraphe 1.2.4.1. Des recherches sont menées sur la gestion des défaillances dans les workflows [94, 100]. Plusieurs solutions sont alors envisagées en réponse à la défaillance d'un travail comme le recouvrement arrière ou l'exécution redondante de travaux. Ces études sont complémentaires avec les travaux que nous présentons.

### 3.1.2 Le système de grille XtreemOS

XtreemOS est un projet européen dont l'objectif est de fournir un *système d'exploitation de grille* capable de gérer des organisations virtuelles. XtreemOS est considéré comme un *système d'exploitation de grille* car il fournit un ensemble complet de services coopérants visant à abstraire les ressources et gérer leur partage entre de multiples utilisateurs. XtreemOS fournit aux applications de grille une interface de type SAGA et une interface POSIX.

L'architecture générale de XtreemOS est présentée sur la figure 3.1. XtreemOS peut intégrer trois types de ressources : des serveurs[1], des grappes de calcul et des appareils mobiles. Une version adaptée à la grille de Linux, appelée XtreemOS-F, est fournie pour

[1]Nous appelons « serveur », une machine physique comportant un petit nombre d'unités de traitement, comme les ordinateurs de bureau

ces trois types de ressources. À noter que LinuxSSI est fondé sur Kerrighed et offre donc une vision de type système à image unique des grappes de calcul.

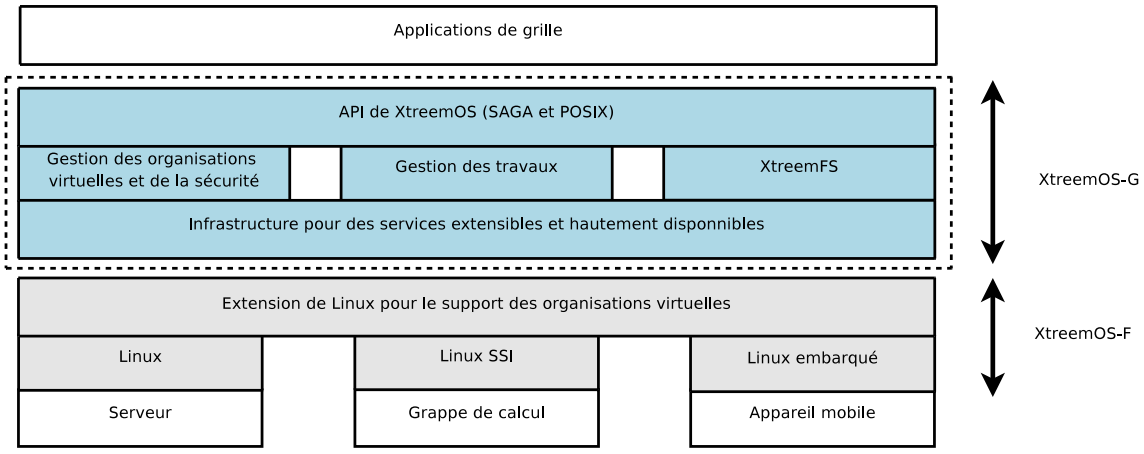



FIG. 3.1 – Architecture de XtreemOS

La couche regroupant l'ensemble des intergiciels est appelée XtreemOS-G. Voici les fonctionnalités fournies par XtreemOS-G :

**Infrastructure pour des services passant à l'échelle et hautement disponibles.** Cette couche offre un ensemble de solutions pour traiter les problèmes liés à la taille et à la versatilité des nœuds de la grille : une solution de réplication active pour des processus multi-threads[2] (*Virtual Nodes* [61]), une solution permettant de déplacer une connexion TCP d'un serveur à un autre (*Distributed Servers* [187]), ainsi que différentes solutions fondées sur des architectures pair-à-pair pour traiter de manière distribuée de grandes quantités de données [170].

**Gestion des organisations virtuelles et de la sécurité.** C'est une boîte à outil pour le support des organisations virtuelles et de la sécurité [49]. Comme il n'existe pas de définition unique d'une organisation virtuelle, l'objectif n'est pas d'imposer un modèle mais d'offrir un ensemble de solutions permettant de configurer l'organisation virtuelle en fonction des besoins.

**Gestion des travaux.** Le service de gestion des travaux (*Application Management Service ou AEM*) doit optimiser l'exécution des travaux sur la grille en dépit de l'hétérogénéité et du dynamisme du système [47]. Pour cela, un gestionnaire de travaux (*job manager*) est responsable pour un travail. Il doit sélectionner des ressources pour ce travail et contrôler le bon déroulement de l'exécution. Il sert aussi de point de contact pour le travail. Sur chaque nœud de la grille, un gestionnaire d'exécution (*execution manager*) a la charge des processus s'exécutant sur ce nœud et applique les décisions prises par le gestionnaire de travaux. Pour gérer les ressources de la grille, et notamment allouer les ressources aux travaux, un gestionnaire de ressources (*resource manager*) et un gestionnaire de réservations (*reservation manager*) sont utilisés. Enfin un annuaire de travaux (*job directory*) permet de localiser le gestionnaire de travaux d'un travail.

**XtreemFS.** XtreemFS [91] est un système de fichier distribué accessible depuis tous les nœuds de la grille. Il permet d'agréger les capacités de stockage réparties sur la grille.

[2]Un *thread*, ou processus léger, est un fil d'exécution d'un processus.

XtreemFS offre la possibilité d'accéder en parallèle à un fichier en répartissant ce fichier sur plusieurs unités de stockage (*striping*). Il offre aussi des mécanismes de réplication cohérente des données, nécessaires à la mise en œuvre d'un support de stockage stable.

**API (SAGA et POSIX).** XtreemOS fournit deux interfaces pour les applications. SAGA (*A Simple API for Grid Applications*) permet de décrire des applications spécialement adaptées à la grille. L'interface POSIX rend possible l'exécution d'applications qui n'étaient pas initialement prévues pour s'exécuter sur grille.

Nous voulons ajouter au module de gestion des travaux de XtreemOS la capacité de traiter les cas de défaillances d'applications en fournissant un service mettant en œuvre des techniques de recouvrement arrière. Cependant comme nous le décrivons dans le paragraphe suivant, ces techniques ont un intérêt dans une grille au delà de la seule tolérance aux fautes.

### 3.1.3 Cas d'utilisation des techniques de recouvrement arrière dans une grille

Les techniques de recouvrement arrière offrent la possibilité de redémarrer une application distribuée à partir d'un point de reprise. Dans une grille, c'est une technique qui peut être utilisée non seulement pour la tolérance aux fautes mais aussi dans d'autres situations. Nous distinguons trois manières d'utiliser des techniques de recouvrement arrière dans la grille :

**Sauvegarde d'un point de reprise d'un travail.** Un point de reprise du travail est sauvegardé pour pouvoir redémarrer le travail depuis ce point plus tard, par exemple en cas de défaillance.

**Migration d'un travail.** La migration d'un travail est son déplacement vers d'autres nœuds de la grille. Pour cela, il faut prendre un point de reprise du travail, arrêter le travail et le redémarrer à partir du point de reprise sur les nouveaux nœuds.

**Suspension d'un travail.** Pour suspendre un travail, il faut prendre un point de reprise du travail et l'arrêter. L'exécution de celui-ci pourra être relancée plus tard à partir du point de reprise, sur les mêmes nœuds ou sur des nœuds différents.

Ces trois utilisations peuvent ensuite être appliquées dans différentes situations :

**Ordonnancement des travaux.** La migration de travaux est utilisée pour optimiser l'ordonnancement des travaux. Un travail peut être déplacé pour équilibrer la charge entre les nœuds de la grille ou pour permettre à d'autres travaux de s'exécuter. Par exemple, si de nouvelles ressources sont intégrées à la grille, des travaux peuvent être déplacés des nœuds les plus chargés de la grille vers ces nouveaux nœuds. Un travail peut aussi être suspendu pour permettre à un travail de plus haute priorité[3] de s'exécuter et reprendre après.

**Déconnexion de nœuds.** Dans le cas d'une déconnexion de nœuds, le propriétaire des ressources demande une autorisation avant de les déconnecter. Il est ainsi possible de déplacer les travaux s'exécutant sur ces nœuds avant d'autoriser la déconnexion.

[3]La notion de priorité d'un travail dépend principalement du modèle d'organisation virtuelle utilisé. L'organisation virtuelle peut définir des priorités sur les travaux selon différents critères, économiques ou autres.

**Tolérance aux fautes.** C'est le cas que nous avons déjà décrit. Des points de reprise du travail sont sauvegardés durant l'exécution normale de l'application pour, en cas de défaillance, ne pas avoir à redémarrer le travail depuis zéro. Des travaux s'intéressent à la tolérance aux fautes pro-active [136]. Il s'agit alors d'analyser l'état courant du système pour tenter de prévoir les défaillances et ainsi déplacer les travaux préventivement. Ceci permet notamment de limiter la fréquence de sauvegarde de points de reprise. Cependant, comme il est impossible de prévoir toutes les fautes, les techniques de tolérance aux fautes classiques sont toujours nécessaires.

**Débogage.** Une autre utilisation possible de la sauvegarde de points de reprise est le débogage des applications [180], dans le cas de défaillances logicielles. Il est ainsi possible de rejouer l'exécution pour tenter de comprendre les raisons de la défaillance.

Ces nombreux cas d'utilisation soulignent l'intérêt d'un service de recouvrement arrière pour la grille.

### 3.1.4 Objectifs

Nous avons présenté dans le chapitre 1 les caractéristiques des grilles de calcul et identifié au début du chapitre 2 les principaux objectifs que nous poursuivons, c'est-à-dire auto-réparation, transparence, passage à l'échelle et performance. Le service a pour objet de rendre le système de grille auto-réparant pour les applications. L'auto-réparation du service en lui-même est traité dans le chapitre 4. Nous raffinons maintenant les autres objectifs dans le contexte d'un service de recouvrement arrière.

**Supporter un grand nombre de travaux.** Le nombre de travaux pouvant s'exécuter au même moment sur la grille étant grand, le service doit être capable de supporter un grand nombre de travaux.

**Prise en charge de différents protocoles.** Plusieurs protocoles de recouvrement arrière existent, offrant différentes propriétés. L'architecture de notre service doit être suffisamment générique pour pouvoir mettre en œuvre des protocoles fondés sur des approches coordonnées ou non coordonnées pour pouvoir choisir le protocole le plus adapté en fonction des caractéristiques de l'application.

**Transparence pour les applications.** Nous voulons être capables de gérer des applications qui n'ont pas initialement été conçues pour fonctionner avec le service sans modification de celles-ci. De plus, nous voulons libérer les concepteurs d'applications de la charge du traitement des défaillances. Ceci implique des mécanismes de sauvegarde de points de reprise transparents pour les applications. Le service doit bien sûr permettre aussi d'exécuter des applications ayant leurs propres mécanismes de tolérance aux fautes.

**Prise en charge de l'hétérogénéité des ressources.** Les nœuds d'une grille sont hétérogènes et peuvent fournir des outils de sauvegarde de points de reprise de processus différents. Par exemple, une grappe de calcul exploitée avec LinuxSSI va fournir un outil de sauvegarde de points de reprise différent d'un serveur exploité avec Linux. Il est possible que pas ne soient disponibles assez de nœuds identiques pour les besoins d'un travail, notamment si ce travail est de grande taille. Le travail pourra alors être distribuée sur des nœuds hétérogènes. XtreemGCP doit être capable d'assurer la tolérance aux fautes de tels travaux.

**Simplicité d'utilisation.** Nous voulons que notre service soit capable de prendre tout seul les décisions de sauvegarde de points de reprise, de gérer les données générées par la sauvegarde des points de reprise et de communiquer avec les autres services

du système, notamment le service de gestion de ressource pour le redémarrage d'un travail depuis un point de reprise. Ainsi l'utilisateur peut ne pas se soucier des problématiques liées à la tolérance aux fautes. Cependant, pour une meilleure qualité de service, nous voulons aussi offrir à l'utilisateur la possibilité de définir ses besoins en tolérance au fautes en définissant par exemple le protocole à utiliser pour un travail donné.

### 3.1.5 Travaux apparentés

Dans ce paragraphe, nous commençons par détailler les techniques existantes pour sauvegarder l'état d'un processus. Puis nous nous intéressons aux techniques de recouvrement arrière pour les applications distribuées, et plus particulièrement aux travaux menés dans le contexte de MPI. Enfin, nous dressons un état des lieux de la gestion des défaillances d'applications dans les systèmes de grille existants.

#### 3.1.5.1 Techniques de sauvegarde de points de reprise de processus

Pour sauvegarder l'image d'un processus, il existe trois principales approches : l'approche niveau applicatif, l'approche niveau utilisateur et l'approche système. Dans les approches niveau applicatif [34, 197], le code permettant la sauvegarde de points de reprise est directement inclus dans l'application. Elles ont l'avantage de minimiser la taille des données sauvegardées pour les points de reprise puisque les mécanismes de sauvegarde des point de reprise sont alors spécifiques à l'application. De plus, elles ont l'avantage d'être indépendantes du matériel. Cependant ces techniques nécessitent une instrumentation du code de l'application, complexifiant la tâche des développeurs. Ceci peut aussi être fait par un pré-compilateur avec l'aide du développeur qui doit ajouter des instructions à son programme pour permettre la sauvegarde de points de reprise, par exemple spécifier à quel moment l'application peut être sauvegardée.

Les approches au niveau utilisateur comme Libckpt [150], Condor [119], DejaVu [164] ou DMTCP [13] se font par l'intermédiaire d'une librairie qui doit être liée à l'application. Ainsi le développeur de l'application n'a pas besoin de s'occuper des problèmes de tolérance aux fautes. Ici c'est l'image du processus du point de vue du système d'exploitation qui est sauvegardée. Cependant les approches au niveau utilisateur ne permettent pas, en général, au redémarrage de rétablir complètement l'état du processus. Par exemple, il n'est pas possible de forcer le redémarrage du processus avec le même identifiant (*PID*). Pour ce problème spécifique, DMTCP introduit la notion d'identifiant virtuel en interceptant l'appel système *fork()* chargé de la création des processus.

Les approches niveau système sont transparentes pour l'application car mises en œuvre par le noyau du système d'exploitation. Elles peuvent nécessiter une modification du noyau [88] ou être mises en œuvre sous forme de module. BLCR [99] (*Berkeley Lab Checkpoint/Restart*) est actuellement la solution au niveau système la plus utilisée. Mise en œuvre sous forme d'un module noyau, elle permet de sauvegarder des processus multithreads et de restaurer au redémarrage les identifiants associés au processus comme le *PID*. Cependant, si ce *PID* est déjà utilisé par un autre processus, le redémarrage échoue. C'est pourquoi des solutions alternatives comme ZAP [108], MCR [81] ou OpenVZ [143] virtualisent les espaces de nommage pour éviter les conflits sur les identifiants au redémarrage. BLCR offre aussi des fonctionnalités au niveau utilisateur. Il permet de définir des fonctions de rappel à exécuter au moment de la sauvegarde d'un point de reprise, au moment du redémarrage de l'exécution après une sauvegarde ou au moment de redémarrage de l'exécution après une défaillance. Il est aussi possible de définir des sections critiques dans

le code de l'application, sections pendant lesquelles un point de reprise ne peut être sauvegardé. La principale limite des approches niveau système est la portabilité. En effet, un processus ne peut, en général, être redémarré qu'avec la même version de l'outil que celle utilisée lors de la sauvegarde.

Les technologies de virtualisation offrant la possibilité de suspendre une machine virtuelle comme Xen [17] ou VMware [203], peuvent être vues comme une solution pour sauvegarder un point de reprise d'un processus. Cependant, ces solutions sont coûteuses car l'image sauvegardée contient alors non seulement l'état du processus mais aussi l'état du système d'exploitation exécuté dans la machine virtuelle.

Ce rapide état de l'art met en évidence la diversité des solutions pour sauvegarder l'état d'un processus. Il montre aussi qu'aucune de ces solutions n'est optimale. Nos objectifs de transparence et de simplicité nous portent vers les solutions mises en œuvre au niveau utilisateur ou au niveau système. Entre ces deux types de solution, celles au niveau système semblent plus attrayantes car elles permettent de sauvegarder et de restaurer plus de détails sur l'état d'un processus. Cependant ces solutions ont aussi leurs limites. Tout d'abord, la nécessité de redémarrer le processus avec la même version de l'outil que pour la sauvegarde peut limiter les possibilités de déplacer les applications dans la grille. De plus, l'approche transparente implique de sauvegarder l'état complet d'un processus alors qu'une approche au niveau applicatif permet de sélectionner les informations nécessaires au redémarrage du processus, réduisant ainsi la quantité de données à sauvegarder. C'est pourquoi nous voulons que XtreemGCP soit capable de prendre en charge toutes ces approches.

#### 3.1.5.2 Techniques de sauvegarde de points de reprise d'applications distribuées

Nous avons abordé les problèmes relatifs à la sauvegarde de points de reprise d'applications distribuées dans le paragraphe 1.2.4.1. En plus de sauvegarder l'état de chaque processus, il faut aussi être capable de prendre en compte l'état des canaux de communication entre les processus pour pouvoir rétablir l'application dans un état cohérent lors d'un retour arrière. Pour cela, un protocole doit être mis en place entre les processus de l'application.

Prendre en charge des problématiques de systèmes distribués au niveau noyau qui, par définition ne connaît que la machine physique sur laquelle il s'exécute, n'est pas une bonne approche [99]. En effet, cela impliquerait de maintenir au niveau noyau des informations sur des entités distantes. C'est pourquoi ceci est fait au niveau utilisateur.

La solution de base pour développer des applications distribuées est d'utiliser l'interface POSIX *socket* permettant à deux processus distribués de communiquer. DejaVu ou DMTCP mettent en œuvre un protocole de sauvegarde de points de reprise coordonnés directement au-dessus de *sockets* TCP en interceptant quelques appels système. Cependant le protocole présenté par DejaVu [164] ne traite pas correctement le cas des processus orphelins en ne considérant pas le cas où de nouveau évènements non déterministes peuvent modifier le comportement des processus au redémarrage. De son côté, DMTCP, dont le protocole nécessite sept synchronisations globales de tous les processus [13] pour sauvegarder un état global cohérent, n'est pas bien adapté aux grilles de calcul où la latence entre certains nœuds peut être élevée.

Des interfaces de plus haut niveau peuvent être mises en œuvre au-dessus de l'interface *socket* pour simplifier la tâche du programmeur. Dans le domaine des applications distribuées à échange de messages, l'interface MPI [71] (Message Passing Interface) s'est imposée comme un standard.

De nombreux travaux ont été menés pour fournir des bibliothèques MPI tolérantes aux défaillances. La plupart des projets, *i.e.* CoCheck [181], Starfish [3], MPI-FT [121], MPICH-V [30], LAM/MPI [166] et Open MPI [93], ont choisi de mettre en œuvre des techniques de recouvrement arrière pour traiter les défaillances de processus. Ici la gestion des canaux de communication est faite au sein de la bibliothèque MPI et un outil de sauvegarde de processus au niveau noyau est utilisé pour sauvegarder l'image des processus. LA-MPI [15] a pour but d'offrir une couche de communication tolérante aux défaillances mais ne supporte pas les défaillances de processus. Enfin FT-MPI [67] propose de modifier la sémantique de MPI pour permettre aux applications de continuer leur exécution en dépit des processus défaillants.

L'approche fondée sur le recouvrement arrière est celle qui nous intéresse car c'est la seule capable de redémarrer les processus défaillants. Les travaux sur LAM/MPI ont intégré l'outil de sauvegarde de points de reprise BLCR. LAM/MPI utilise les fonctions de rappel de BLCR pour mettre en œuvre un protocole de sauvegarde de points de reprise coordonnés. LAM/MPI a adopté une approche modulaire. Ainsi, contrairement aux autres solutions citées plus tôt, il n'est pas fortement couplé à BLCR et peut intégrer d'autres outils de sauvegarde de processus. Open MPI a repris cette approche modulaire [92] et a pour le moment intégré deux modules : un fondé sur BLCR et un, nommé *self*, permettant de mettre en œuvre une sauvegarde de processus au niveau applicatif. Contrairement à LAM/MPI qui est fondé sur la sauvegarde de points de reprise coordonnés, Open MPI propose aussi une approche modulaire pour ce qui concerne les protocoles de recouvrement arrière, offrant ainsi la possibilité de mettre en œuvre différents protocoles : coordonnés ou non coordonnées, avec ou sans enregistrement de messages.

#### 3.1.5.3 Recouvrement arrière dans les grilles de calcul

Le recouvrement arrière est le plus souvent abordé dans le cadre de la tolérance aux fautes dans les grilles, quand bien même d'autres cas d'utilisation existent comme nous l'avons décrit dans le paragraphe 3.1.3. C'est donc sous cet angle que sont comparés les travaux décrits dans ce paragraphe. Nous pensons que cette approche est pertinente car la tolérance aux fautes est le cas d'utilisation le plus complexe à traiter. En effet, c'est le seul cas où le recouvrement arrière intervient en réaction à un évènement non prévu, une défaillance.

Les travaux de l'OGF sur les problèmes de tolérance aux fautes dans les grilles de calcul ont conduit à la spécification de l'architecture d'un service de sauvegarde de points de reprise dans la grille, nommée *GridCPR* [183]. Ce travail est limité aux travaux exécutées sur un seul nœud de la grille et ne considère que les techniques de sauvegarde de points de reprise au niveau applicatif. Les travaux préliminaires sur les APIs de ce service ont été intégrées à SAGA sous forme de quatre appels : *suspend()*, *resume()*, *checkpoint()* et *migrate()*. Enfin ces travaux mettent en avant deux points cruciaux relatifs à la conception d'un service de recouvrement arrière dans la grille : la gestion des données générées par les mécanismes de sauvegarde de points de reprise et le redémarrage automatique des travaux après une défaillance. Avant d'aller plus loin, détaillons ces deux points :

**Redémarrage automatique des travaux** Redémarrer automatiquement un travail nécessite de trouver des ressources disponibles compatibles avec les besoins de ce travail. Il est donc nécessaire d'interagir avec le service de gestion des ressources. Cela implique aussi de connaître les besoins de ce travail comme les besoins en ressources de calcul ou le temps d'exécution, qui ont normalement été spécifiés lors de la soumission du travail. Il faut donc sauvegarder ces informations en plus de

l'état du travail pour être capable de le redémarrer correctement.

**Gestion des données de points de reprise** Comme nous l'avons vu précédemment, les données nécessaires au redémarrage d'un travail sont nombreuses. Elles comprennent l'état des processus, des informations sur les canaux de communications, les besoins du travail, *etc*. Gérer toutes ces données dans un environnement distribué comme une grille est une tâche complexe. Il faut tout d'abord être capable de localiser toutes ces informations. Il faut ensuite pouvoir les rendre disponibles sur les nœuds sélectionnés pour le redémarrage. Enfin la gestion de l'espace de stockage alloué à la sauvegarde de ces informations est une tâche à ne pas négliger étant donné la quantité de données concernées.

Dans le contexte du réseau d'excellence CoreGRID [138], un service de sauvegarde de points de reprise, nommé GCA [98, 97] (*Grid Checkpointing Architecture*), a été proposé. Celui-ci peut intégrer plusieurs outils de sauvegarde de points de reprise à la grille à l'aide d'un service nommé *Checkpoint Translation Service*, chargé de faire l'interface entre les services de grilles et les outils locaux de sauvegarde de points de reprise. La principale solution envisagée pour la sauvegarde des points de reprise est l'utilisation de machines virtuelles. Un premier prototype de ce service [96] intègre un outil de sauvegarde de points de reprise niveau noyau pour système Altix. La principale limite du service est qu'il ne peut gérer que des travaux s'exécutant sur un seul nœud. De plus, il est fondé sur une architecture centralisée et ne considère que des grappes de calcul exploitées via un gestionnaire de traitement par lot. Dans ce cadre, la localisation des données générées par la sauvegarde des points de reprise et le redémarrage automatique des applications après une défaillance est pris en charge.

HPC4U [90] est un projet intégrant la notion de qualité de service dans les grilles. Dans ce cadre a été proposée une solution de sauvegarde de points de reprise pour la tolérance aux fautes [78] utilisant l'outil de sauvegarde de points de reprise niveau système MCR. Le système est aussi capable de sauvegarder des applications MPI utilisant la bibliothèque propriétaire Scali MPI. L'approche retenue ici est la sauvegarde de points de reprise coordonnés. Une limitation du système est que les travaux ne peuvent pas être déplacés entre des domaines d'administration différents à cause des limites du système de gestion des ressources. Le redémarrage des applications se fait donc au sein du même domaine d'administration.

InteGrade [55] est un intergiciel de grille, visant plus particulièrement les réseaux de station de travail. Il inclut un service de sauvegarde de points de reprise capable de sauvegarder et de redémarrer automatiquement des applications parallèles de type BSP [194] (*Bulk Synchronous Parallel*). La mise en œuvre de la sauvegarde des points de reprise est faite au niveau applicatif.

Migol [123] étend les services de Globus [5] pour assurer la bonne terminaison des applications dans la grille en dépit des défaillances. Pour cela, Migol se base sur des mécanismes de sauvegarde de points de reprise au niveau applicatif. Les applications MPI sont supportées mais nécessitent d'être liées avec une bibliothèque spécifique fournie par Migol. En cas de défaillance, Migol est capable de sélectionner les ressources les plus appropriées sur la grille pour le redémarrage de l'application. Enfin, Migol propose des stratégies de réplication des données pour éviter la perte de points de reprise.

Condor-G [77] étend lui aussi Globus notamment pour fournir de la tolérance aux fautes aux applications. Pour cela, l'outil de sauvegarde de points de reprise de Condor est utilisé. Ainsi des travaux simples peuvent être sauvegardés. Condor-G est capable de redémarrer automatiquement un travail défaillant. Cependant les problèmes associés au traitement des données de sauvegarde de points de reprise ne sont pas évoqués.

DIET [8] est fondé sur le modèle client-serveur au travers du modèle Grid-RPC. Des mécanismes de sauvegarde de points de reprise sont utilisés pour assurer la tolérance aux fautes des serveurs [29]. DIET peut exploiter des outils de sauvegarde de points de reprise au niveau utilisateur tel que Condor et peut aussi gérer le cas de mécanismes mis en œuvre au niveau applicatif. Pour traiter le cas d'applications distribuées et notamment d'applications MPI, DIET se fonde sur l'utilisation de bibliothèques tolérantes aux fautes telles que MPICH-V. JuxMEM [14] est utilisé pour sauvegarder les points de reprise de façon stable. Cependant le problème de la gestion de ces données n'est pas évoqué. Le redémarrage d'un serveur défaillant est à la charge du client utilisant ce serveur au moment de la défaillance. NetSolve [4] est un autre système de grille fondé sur le modèle client-serveur. NetSolve à une approche comparable à DIET mais se fonde sur Starfish pour la tolérance aux fautes d'applications distribuées.

Open MPI ne traite pas les problèmes liés à l'exécution sur grille. Cependant, il est important de mentionner les travaux effectués sur la représentation des points de reprise d'applications distribuées et la gestion de ses données. Ainsi Open MPI inclus toutes les informations nécessaires au redémarrage de l'application dans le fichier représentant le point de reprise comme les paramètres utilisés pour l'exécution de l'application ou la localisation de fichier représentant l'état de chaque processus.

| | Applications supportées | Sauvegarde des processus | Gestion des données de tolérance aux fautes | Redémarrage Automatique |
|---|---|---|---|---|
| GridCPR [183] | Travaux simples non distribués | Applicatif | Évoqué | Évoqué |
| GCA [97] | Travaux simples non distribués | **Générique** | Partiel | **Oui** |
| HPC4U [78] | Applications Scali MPI | Système | Partiel | Partiel |
| InteGrade [55] | Applications parallèles BSP | Applicatif | Non | **Oui** |
| Migol [123] | **Applications MPI** | Applicatif | Partiel | **Oui** |
| Condor-G [77] | Travaux simples non distribués | Utilisateur | Non | **Oui** |
| DIET [29] | **Applications Distribuées** | Utilisateur ou Applicatif | **Oui** | **Oui** |
| Open MPI [93] | **Applications MPI** | **Générique** | Partiel | Non |
| XtreemGCP | **Applications Distribuées** | **Générique** | **Oui** | **Oui** |

Tab. 3.1 – Tableau de comparaison des outils de recouvrement arrière dans les grilles

Le tableau 3.1 résume les principales caractéristiques des travaux que nous avons présentés. On peut voir que les travaux qui se rapprochent le plus de nos objectifs sont ceux de Open MPI et DIET. Cependant Open MPI ne résout pas tous les problèmes liés à l'exécution sur grille. De plus, par définition, Open MPI ne traite que le cas des applications MPI. DIET est lui dépendant de bibliothèques telles que Open MPI pour sauvegarder des points de reprise d'applications distribuées.

Le service GCA est aussi intéressant car il utilise une interface générique pour les

outils de sauvegarde de points de reprise, évitant ainsi d'être dépendant d'un outil particulier. Cependant GCA ne considère pas les applications distribuées sur plusieurs nœuds de la grille. La plupart des solutions mentionnées prennent bien en compte les problèmes liés au redémarrage automatique des applications. La gestion des données de points de reprise est beaucoup moins bien traitée. Si la plupart assurent l'accessibilité des données au redémarrage, ils ne prennent pas en compte les problèmes liés à la gestion de l'espace de stockage, c'est-à-dire définir qui fournit cet espace de stockage ou définir quand les données peuvent être supprimées.

Notre objectif dans XtreemGCP est résumé sur la dernière ligne du tableau 3.1. Nous voulons être capables de couvrir un large spectre d'applications distribuées, incluant les applications MPI. Ceci signifie nous voulons être capables de tirer partie d'approches telles que Open MPI mais que nous voulons aussi offrir une solution de tolérance aux fautes directement au dessus de *sockets* TCP. Nous visons une approche générique permettant d'intégrer différentes solutions de sauvegarde de processus. Enfin nous voulons que XtreemGCP soit capable de gérer les données générées par la sauvegarde de points de reprise et soit capable de redémarrer automatiquement les applications défaillantes.

### 3.1.6 Conclusion

L'évaluation de l'état de l'art, au regard de nos objectifs, met en évidence le besoin de concevoir un nouveau service pour mettre en œuvre des techniques de recouvrement arrière dans la grille et en particulier dans XtreemOS.

Ce service doit fournir des mécanismes de recouvrement arrière pour les travaux s'exécutant sur la grille. Ces travaux peuvent être exécutés sur un seul nœud de la grille, pour les plus simples, ou être des applications parallèles distribuées sur plusieurs nœuds. Dans ce cas, nous considérons un modèle simple où les processus communiquent en s'échangeant des messages.

L'évaluation des techniques de sauvegarde de l'état d'un processus montre qu'aucune solution n'est parfaite : les techniques au niveau applicatif sont les plus efficaces mais ne sont pas transparentes pour l'application ; les techniques au niveau système sont transparentes mais la taille de l'état sauvegardé est alors plus important. De plus, un processus sauvegardé avec un outil système ne peut en général être redémarré qu'avec la même version de cet outil, limitant à priori la portabilité d'une telle solution dans la grille. Cependant l'approche de XtreemOS est différente de celle des autres projets de grille puisqu'il vise à fournir un système d'exploitation complet adapté à la grille. Ce système peut donc intégrer des outils de sauvegarde de points de reprise par défaut, résolvant ainsi les problèmes de portabilité. C'est pourquoi nous privilégions l'approche système qui permet d'atteindre l'objectif de transparence. Toutefois nous ne voulons pas lier notre service à un outil de sauvegarde de points de reprise particulier. C'est pourquoi nous voulons définir une interface standard entre le service et ces outils.

Dans le cas d'applications parallèles, la gestion des canaux de communication se fait toujours au niveau utilisateur ou au niveau applicatif. En général, la conception d'applications parallèles se fait avec l'aide d'une bibliothèque dédiée. MPI est alors l'interface la plus couramment utilisée. Les bibliothèques MPI fournissant des mécanismes de tolérance aux fautes se chargent des canaux de communication et utilisent un outil de sauvegarde au niveau système pour les processus. L'approche modulaire d'Open MPI lui permet d'utiliser différents outils pour la sauvegarde de points de reprise de processus. Les travaux de LAM/MPI avec BLCR ont montré l'intérêt de fonctions de rappel exécutées dans leur propre fil d'exécution au moment de la sauvegarde et de la restauration des processus pour

gérer les canaux de communication. Nous proposons d'utiliser ces fonctions de rappel pour mettre en œuvre des protocoles de sauvegarde de points de reprise directement au niveau de l'API *socket* en mode TCP et ainsi de fournir des protocoles par défaut aux utilisateurs de la grille.

Enfin, dans le contexte d'XtreemOS, le service de recouvrement arrière doit être capable d'interagir avec les autres services de grille, et tirer avantage de ceux-ci. Des interactions avec l'AEM sont nécessaires pour le redémarrage automatique des applications mais aussi pour traiter les autres cas d'utilisation comme la migration de travaux. XtreemFS fournit un support de stockage stable accessible depuis tous les nœuds de la grille. Étant donné les quantités de données que peuvent générer les mécanismes de recouvrement arrière, il est nécessaire de définir des règles simples pour gérer l'espace de stockage nécessaire au fonctionnement du service de recouvrement arrière.

## 3.2 XtreemGCP : un service de traitement automatique des défaillances fondé sur le recouvrement arrière pour des applications distribuées

Dans ce chapitre, nous commençons par présenter l'architecture de service XtreemGCP. Puis nous décrivons l'interface générique permettant l'utilisation de différents outils de sauvegarde de points de reprise de processus. Nous détaillons ensuite la sauvegarde de points de reprise et le redémarrage d'application distribuées par XtreemGCP. Enfin, après avoir décrit la politique de gestions des données de point de reprise, nous présentons l'interface fournit aux utilisateurs du service.

### 3.2.1 Définitions

Avant d'entrer dans les détails du service, nous définissons le vocabulaire utilisé. Comme nous l'avons déjà présenté, un **travail** est l'exécution sur la grille d'une application décrite par un fichier JSDL. Cette application peut être exécutée sur un seul nœud ou être une application distribuée fortement couplée, *i.e.* les entités la composant ne peuvent pas être exécutées indépendamment les unes des autres. Dans le cas contraire, on parle d'application distribuée faiblement couplée. Ceci peut correspondre au paradigme maître-travailleurs ou client-serveur, à une application de couplage de code ou encore à un *workflow*. Dans ce cas les entités distribuées sont des travaux différents. Une **application de grille** gère alors cet ensemble de travaux.

Un travail distribué sur plusieurs nœuds de la grille est composé d'**éléments**. Un processus d'un travail s'exécutant sur un nœud de la grille peut créer des processus fils. On parle alors d'arborescence de processus. Un élément est une arborescence de processus. On peut envisager avoir plusieurs éléments d'un travail s'exécutant sur un nœud, en particulier si ce nœud est une grappe de calcul exploitée avec LinuxSSI.

Enfin dans un soucis de simplification, nous proposons le terme **checkpointeur**[4] pour désigner un outil de sauvegarde de points de reprise.

### 3.2.2 Architecture du système

Lors de la conception de l'architecture de XtreemGCP, deux principales contraintes sont à prendre en compte. Tout d'abord le service doit supporter un grand nombre d'applications. Une architecture centralisée mettrait de fortes contraintes sur le point central

[4]Directement inspiré de l'anglais « *checkpointer* »

et pourrait limiter le passage à l'échelle du service. C'est pourquoi nous voulons éviter que notre service repose sur un point central. La deuxième contrainte est que nous voulons être capable de prendre en charge des travaux distribués sur plusieurs nœuds et être capable d'appliquer des protocoles de tolérance aux fautes entre les éléments distribués de ces travaux.

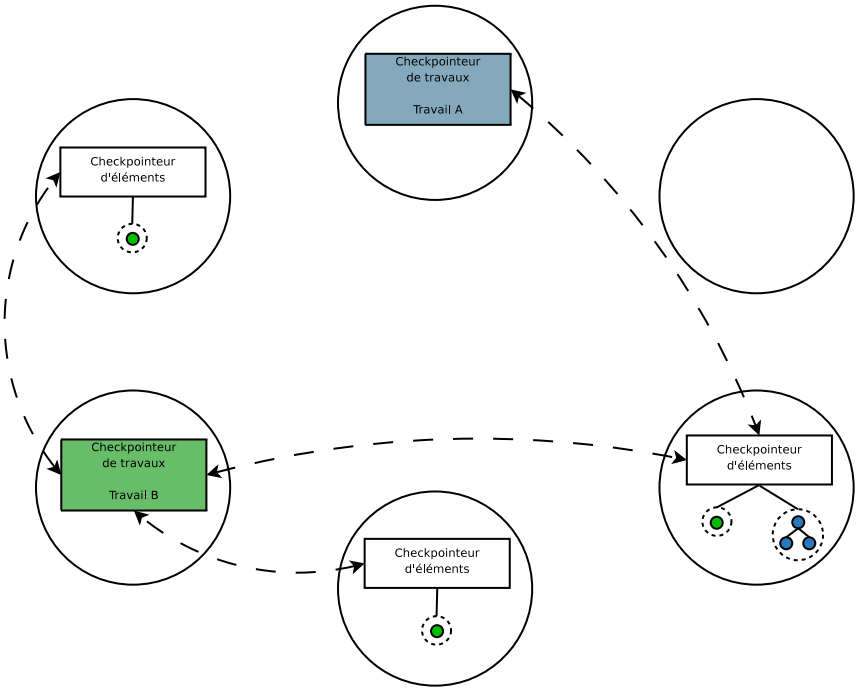



Fig. 3.2 – Architecture de XtreemGCP

L'architecture de notre service est illustrée par la figure 3.2, représentant quelques nœuds d'une grille de calcul. Il est composé de checkpointeurs de travaux et de checkpointeurs d'éléments.

**Checkpointeur de travaux.** Il y a un checkpointeur de travaux par travail. Son positionnement dans la grille est indépendant des nœuds sur lesquels les éléments du travail s'exécutent. Il peut, à priori, être exécuté sur n'importe quel nœud de la grille. Dans le cadre de XtreemOS, nous choisissons de le positionner sur le même nœud que le gestionnaire de travaux en charge de ce travail car les interactions entre ces deux services peuvent être fréquentes. L'annuaire de travaux de XtreemOS peut être utilisé pour retrouver la position d'un checkpointeur de travaux. Le checkpointeur de travaux a la charge d'appliquer les décisions de recouvrement arrière pour un travail. Il synchronise l'action des checkpointeurs d'éléments quand cela est nécessaire.

**Checkpointeur d'éléments.** Sur chaque nœud de la grille, est positionné un checkpointeur d'éléments. Celui-ci commande les checkpointeurs de processus présents sur le nœud pour sauvegarder l'état des éléments s'exécutant sur ce nœud. Il interagit avec les checkpointeurs de travaux des travaux correspondants pour appliquer leurs décisions.

Sur la figure 3.2, deux travaux (A et B) s'exécutent sur la grille. Un checkpointeur de travaux est donc responsable pour chacun de ces travaux. Dans un soucis de clarté, nous n'avons pas ici représenté tous les checkpointeurs d'éléments mais seulement ceux qui ont

un intérêt dans cet exemple. Il y en a normalement un sur chaque nœud. Le travail A est composé d'un seul élément comprenant une arborescence de processus. Le checkpointeur d'éléments responsable pour cet élément va devoir utiliser un checkpointeur de processus capable de gérer des arborescences de processus. Le travail B est une application parallèle distribuée sur plusieurs nœuds de la grille. Des interactions peuvent alors être nécessaires entre le checkpointeur de travaux et les checkpointeurs d'éléments pour assurer que les informations sauvegardées dans un point de reprise constituent un état global cohérent. À noter qu'un checkpointeur d'éléments peut avoir à gérer des éléments de différents travaux.

Nous revenons en détail sur le rôle des deux entités composant notre service dans le paragraphe 3.2.4, décrivant la prise en charge des travaux distribués.

### 3.2.3 Interface générique pour les outils de sauvegarde de points de reprise de processus

L'état de l'art présenté dans le paragraphe 3.1.5.1 a montré que de nombreuses solutions existent pour la sauvegarde de l'état d'un processus. Notre objectif de transparence nous porte vers les solutions mises en œuvre au niveau système comme BLCR et OpenVZ. Cependant nous ne voulons pas négliger les autres types de solutions qui offrent des avantages différents. C'est pourquoi nous reprenons l'idée introduite dans GCA [97] d'une interface générique pour les outils de sauvegarde de points de reprise de processus, ou checkpointeur de processus.

#### 3.2.3.1 Description de l'interface

Notre objectif ici n'est pas d'entrer dans les détails de cette interface mais simplement d'en donner une vision de haut niveau. Voici les principales fonctionnalités que doit fournir cette interface :

- *checkpoint()* : Cet appel attend l'identifiant d'un élément (en général le *PID* du processus père dans l'arborescence) et appelle le checkpointeur de processus pour sauvegarder un point de reprise de cet élément dans un fichier.
- *restart()* : Cet appel restaure un élément à partir d'un fichier décrivant un point de reprise par l'intermédiaire du checkpointeur de processus.
- *resume()* : Cet appel avertit un élément qu'il peut reprendre son exécution normale après avoir été arrêté pour sauvegarder un point de reprise ou après avoir été restauré à partir d'un point de reprise.
- *register_callback()* : Cet appel permet d'enregistrer des fonctions de rappel pour un élément. Ces fonctions sont appelées avant la sauvegarde d'un point de reprise, au moment de la reprise de l'exécution après la sauvegarde d'un point de reprise, après la restauration de l'état des processus lors du redémarrage à partir d'un point de reprise.
- *disable_checkpoint() et enable_checkpoint()* : Ces deux appels permettent de définir des sections critiques dans le code, où un point de reprise ne peut pas être sauvegardé.

Cette simple interface s'inspire de l'interface proposée par BLCR [99]. La possibilité d'enregistrer des fonctions de rappel permet de traiter les informations qui ne peuvent pas directement être traitées par le checkpointeur de processus. Elles sont exécutées dans un fil d'exécution séparé des fils d'exécutions applicatifs. Comme nous allons le voir, elles peuvent notamment être utiles pour traiter les canaux de communication entre processus d'une application distribuée. Une description détaillée de cette interface est disponible dans [46].

#### 3.2.3.2 Intégration d'outils de sauvegarde de points de reprise de processus

Le checkpointeur d'éléments du service XtreemGCP n'est pas lié à un checkpointeur de processus particulier. Il utilise l'interface générique que nous venons de définir. Il peut donc utiliser tout checkpointeur de processus si l'interface générique a été mise en œuvre pour ce checkpointeur. L'utilisation de plusieurs checkpointeurs de processus est illustrée par la figure 3.3. Nous reprenons ici l'exemple du nœud de la figure 3.2 où deux éléments de travaux différents sont exécutés sur le même nœud. Imaginons que ces deux travaux aient des caractéristiques différentes. Dans ce cas, il est possible que les checkpointeurs de processus les mieux adaptés à chacun de ces éléments soient différents. L'interface générique permet au checkpointeur d'éléments d'utiliser simplement différents checkpointeurs de processus. Ainsi, sur la figure 3.3, le checkpointeur de processus *A* est utilisé pour un élément et le checkpointeur de processus *B* pour l'autre. Dans le paragraphe 3.2.3.3, nous décrivons le processus de sélection d'un checkpointeur de processus.

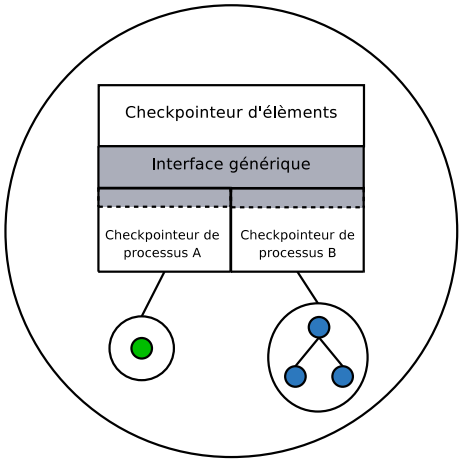



Fig. 3.3 – Utilisation de plusieurs checkpointeurs de processus par XtreemGCP

L'utilisation de cette interface générique offre aussi la possibilité d'intégrer de nouveaux checkpointeurs de processus à XtreemGCP, sans modification du service.

De la même manière, des bibliothèques de communication telles que Open MPI ayant une approche modulaire pour l'intégration de checkpointeurs de processus, peuvent tirer avantage de cette interface générique. En mettant en œuvre un module utilisant notre interface générique, elles pourraient ensuite utiliser de manière transparente les checkpointeurs de processus disponibles sur les nœuds de la grille XtreemOS.

#### 3.2.3.3 Sélection d'un outil de sauvegarde de points de reprise de processus

Pour sélectionner un checkpointeur de processus, il faut pouvoir en connaître les caractéristiques. Pour cela, nous reprenons l'idée proposée par GCA [97]. Lors de la connexion d'un nœud à la grille, les checkpointeurs de processus disponibles sur ce nœud doivent être annoncés. Comme nous l'avons vu dans le paragraphe 3.1.5.1, les checkpointeurs de processus ont des capacités différentes. Par exemple, seuls certains sont capables de sauvegarder et de restaurer l'identifiant d'un processus. C'est pourquoi une description des capacités de chaque checkpointeur doit être fournie. Cette description est ensuite utilisée par le gestionnaire de ressources comme critère pour la découverte de ressources. Ainsi il peut essayer de trouver pour chaque travail des nœuds hébergeant un checkpointeur de processus ayant

les capacités requises par ce travail.

Dans XtreemOS, tous les nœuds pouvant exécuter des travaux sont équipés par défaut d'au moins un outil de sauvegarde de processus au niveau système. Nous considérons ici que les appareils mobiles ne sont pas des ressources disponibles pour l'exécution de travaux et qu'ils peuvent seulement être utilisés comme client. Ainsi, pour la version installée sur les serveurs, XtreemOS fournit BLCR et OpenVZ. LinuxSSI a lui aussi son outil de sauvegarde de processus. Ces checkpointeurs des processus au niveau système sont suffisamment complets pour être capables de sauvegarder l'état des processus s'exécutant sur la grille dans la plupart des cas. Cependant il est toujours possible d'intégrer d'autres checkpointeurs dans un soucis d'optimisation des performances par exemple.

### 3.2.4 Prise en charge des applications distribuées

Appliquer des techniques de recouvrement arrière à des applications distribuées nécessite non seulement d'être capable de sauvegarder l'état de chaque processus, mais aussi de prendre en compte l'état des canaux de communication, pour être capable de redémarrer l'application dans un état global cohérent.

Comme nous l'avons vu dans le paragraphe 3.1.5.2, la gestion des canaux de communication ne se fait pas au niveau système. Nous envisageons alors deux cas. Les canaux de communication peuvent être gérés par la bibliothèque de communication utilisée pour décrire l'application. Nous avons vu que ce type de fonctionnalités est fourni dans les principales bibliothèques MPI. Les mécanismes sont alors mis en œuvre directement au niveau applicatif. Le deuxième cas est celui où l'application n'utilise pas de bibliothèque spécifique pour les communications ou que la bibliothèque utilisée ne fournit pas de mécanismes de recouvrement arrière. Nous proposons alors que XtreemGCP, avec l'aide des mécanismes de fonctions de rappel, prennent en charge la gestion des canaux de communication. En reprenant la classification utilisée pour les checkpointeurs de processus, le mise en œuvre se fait alors au niveau utilisateur. Nous présentons maintenant la prise en charge de ces deux cas par XtreemGCP.

#### 3.2.4.1 Mise en œuvre au niveau utilisateur

La technique de recouvrement arrière la plus couramment utilisée est la sauvegarde de points de reprise coordonnés. Dans ce cas, les processus sont coordonnés avant la sauvegarde du point de reprise pour assurer que l'ensemble des images des processus forment un état global consistant. C'est la solution la plus simple à mettre en œuvre. C'est aussi la solution la plus adaptée pour la migration ou la suspension d'applications. En effet, l'application peut alors être arrêtée dans un état global consistant pour être redémarrée ailleurs et/ou plus tard. C'est pourquoi nous proposons de fournir une solution pour la sauvegarde de points de reprise coordonnés au sein de XtreemGCP.

**Points de reprise coordonnés** Le protocole de sauvegarde de points de reprise que nous utilisons est celui décrit par LAM/MPI [166]. Nous avons choisi ce protocole car l'absence de synchronisation globale de tous les processus pendant la sauvegarde d'un point de reprise en fait un protocole particulièrement adapté aux applications s'exécutant sur grille. Nous le mettons en œuvre dans une bibliothèque qui peut être liée dynamiquement à l'application. Nous utilisons notamment le mécanisme de fonctions de rappel fournit par un checkpointeur de processus tel que BLCR.

La figure 3.4 décrit la sauvegarde d'un point de reprise coordonné pour une application distribuée composée de deux processus. Le checkpointeur de travaux joue alors le rôle de

coordinateur. Pour sauvegarder un point de reprise d'un travail, il commence par notifier les checkpointeurs d'éléments des nœuds sur lesquels s'exécute un élément du travail (étape 1 de la figure). Les checkpointeurs d'éléments appellent *checkpoint()* pour les processus concernés (étape 2). Il faut alors trouver un état global consistant en vidant les canaux de communication entre les processus avant de sauvegarder l'état de chaque processus.

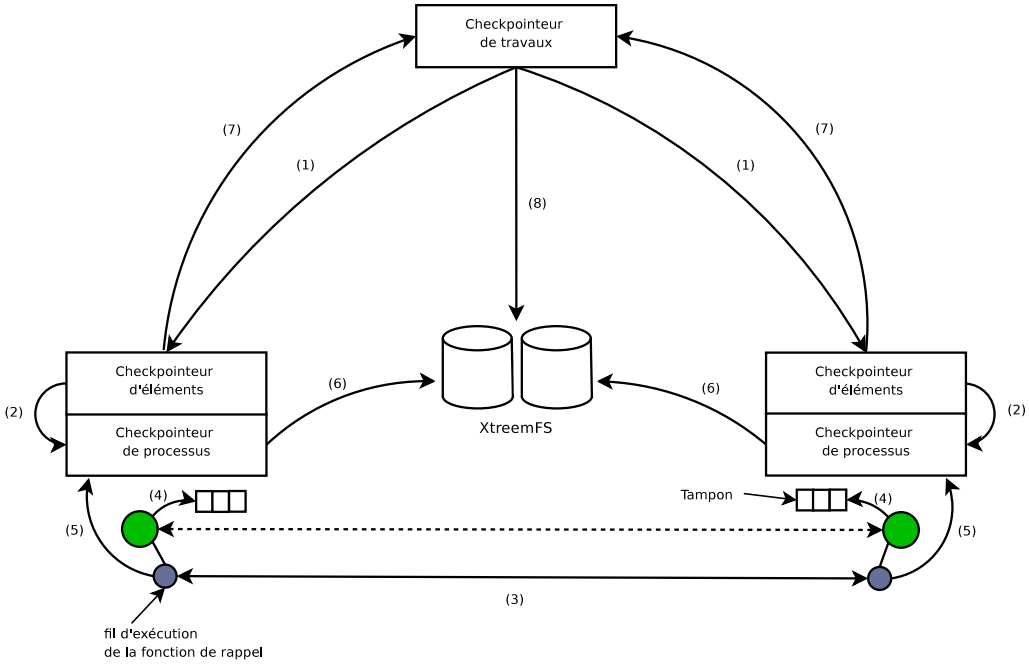



FIG. 3.4 – Étapes de la sauvegarde d'un point de reprise coordonné

Pendant l'exécution normale du travail, la bibliothèque surveille les communications entre ses éléments en interceptant les appels système pouvant servir à l'envoi et à la réception de messages, *i.e. send()*, *recv()*, *read()* et *write()* pour connaître les quantités de données émises et reçues par chaque processus. Pour vider les canaux de communications, il faut empêcher les processus de continuer à envoyer des messages tout en leur permettant de recevoir les dernières données qu'ils n'ont pas encore reçues. L'appel de la fonction *checkpoint()* par le checkpointeur d'éléments signale au checkpointeur de processus qu'il doit commencer la préparation de la sauvegarde du point de reprise. Pour cela, il réveille les fonctions de rappel associées aux éléments à sauvegarder. Parmi ces fonctions de rappel se trouvent une fonction, préalablement enregistrée, qui permet aux processus de s'échanger deux à deux les valeurs des quantités de données qu'ils se sont envoyés (étape 3). Ces valeurs sont comparées avec les quantités de données reçues pour savoir s'il reste des données à recevoir. À partir de ce moment, tous les envois de données d'un processus sont interceptés et les données à envoyer stockées dans un tampon (étape 4). Comme plus aucune donnée n'est envoyée sur les canaux de communication, et que les dernières données en transit continuent d'être reçues, les canaux de communication finissent par être vides. Le checkpointeur de processus est alors averti que l'élément est prêt à être sauvegardé (étape 5). L'exécution de l'élément est alors arrêtée et son état sauvegardé sur support stable, XtreemFS dans notre cas (étape 6). Cet état inclut le tampon dans lequel ont été temporairement stockés les messages à envoyer. Le checkpointeur d'éléments informe alors le checkpointeur de travaux du succès ou de l'échec de la sauvegarde de l'élément

en indiquant le nom du fichier représentant l'image de l'élément (étape 7). Si tous les éléments sont sauvegardés avec succès, le checkpointeur de travaux a alors l'assurance que l'ensemble des images d'éléments sauvegardées forme un état global cohérent. Il peut alors créer le fichier décrivant le point de reprise coordonné contenant la liste des images de chaque élément (étape 8).

Le redémarrage d'un travail à partir d'un point de reprise est décrit par la figure 3.5. Tout d'abord, le checkpointeur de travaux informe les checkpointeurs d'éléments des nœuds sélectionnés pour redémarrer le travail de l'élément qu'ils ont à restaurer (étape 1 sur la figure). Chaque checkpointeur d'éléments récupère alors le fichier de point de reprise de l'élément sur XtreemFS pour connaître le checkpointeur de processus à utiliser pour redémarrer l'élément (étape 2), puis appelle *restart()* pour restaurer l'état des processus composant l'élément (étape 3). Quand ceci est fait, les checkpointeurs d'éléments notifient le checkpointeur de travaux (étape 4) qui attend de savoir que tous les éléments composant le travail ont été restaurés. Quand cette condition est remplie, le checkpointeur de travaux demande aux checkpointeurs d'éléments de redémarrer les éléments. Ce redémarrage par l'appel à *resume()* (étape 6) réveille une fonction de rappel chargée de rétablir les connections entre les processus (étape 7) et d'envoyer les données stockées dans le tampon local lors de la sauvegarde du point de reprise coordonné (étape 8).

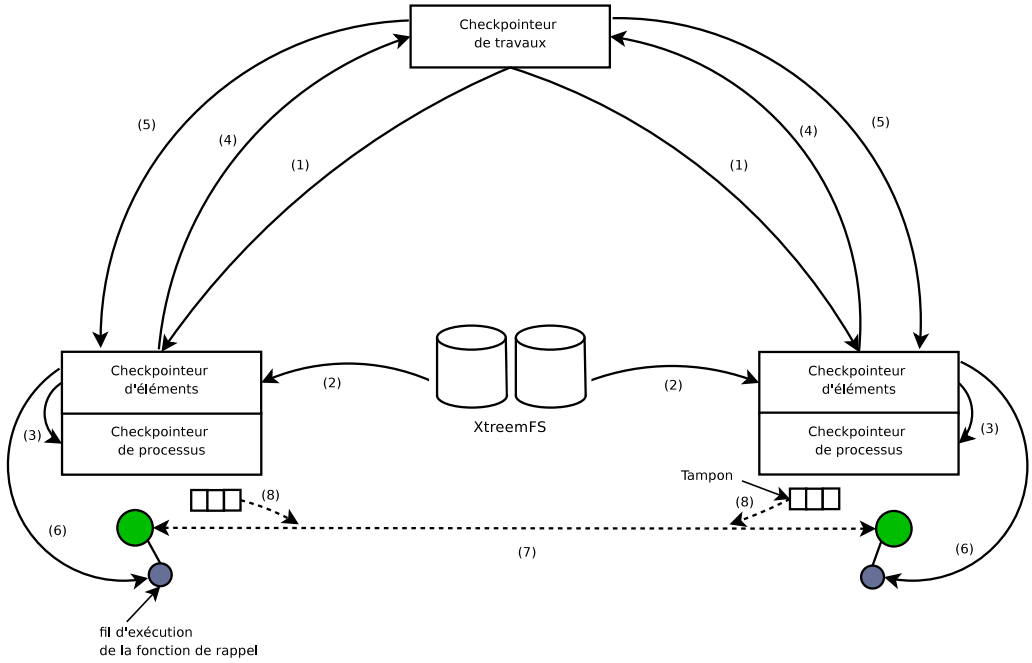



FIG. 3.5 – Étapes du redémarrage à partir d'un point de reprise coordonné

**Autres techniques de sauvegarde de points de reprise** Il est envisageable de mettre en œuvre d'autres techniques de sauvegarde de points de reprise, comme des techniques de sauvegarde de points de reprise non coordonnées, avec ou sans enregistrement de messages, au sein du service XtreemGCP.

Dans ce cas, la sauvegarde des éléments composant le travail est faite indépendamment. De plus, il est alors nécessaire de sauvegarder des informations de dépendances entre les éléments tout au long de l'exécution du travail pour être capable de restaurer l'application

dans un état global consistant au redémarrage. L'architecture de XtreemGCP permet de mettre œuvre ce type de solutions en rendant les checkpointeurs d'éléments responsables d'appliquer le protocole sélectionné pour chaque élément.

Cette problématique sort du cadre de nos travaux. C'est pourquoi nous n'entrons pas plus dans les détails ici. Nos travaux sur la mise en œuvre au niveau applicatif de protocoles de sauvegarde de points de reprise fondés sur l'enregistrement de messages, sont décrits dans la chapitre 5.

#### 3.2.4.2 Mise en œuvre au niveau applicatif

Des mécanismes de sauvegarde de points de reprise peuvent être directement intégrés au sein de l'application distribuée, ou, plus fréquemment être fournis par la bibliothèque de communication utilisée par l'application. Ainsi la plupart des bibliothèques MPI intègrent des solutions de sauvegarde de points de reprise. Il est important que XtreemGCP puisse tirer partie des fonctionnalités fournies par ces bibliothèques. Un checkpointeur d'élément peut alors être désigné responsable pour interagir avec l'application.

Pour illustrer ce point, nous prenons l'exemple d'une application Open MPI. Dans une application Open MPI, c'est le processus qui démarre l'application (*mpirun*) qui a la charge d'appliquer les décisions de sauvegarde de points de reprise. Pour que XtreemGCP soit capable de sauvegarder et redémarrer une application Open MPI, il suffit donc que le checkpointeur d'élément situé sur le nœud où est exécuté le *mpirun* soit capable d'interagir avec celui-ci.

#### 3.2.4.3 Utilisation pour la tolérance aux fautes

Dans le cas de la migration ou de la suspension d'un travail, la décision de sauvegarder un point de reprise du travail est prise par le service de gestion des ressources. Cette décision est ensuite appliquée par le checkpointeur de travaux qui démarre la sauvegarde d'un point de reprise coordonné.

Lorsque les mécanismes de recouvrement arrière sont utilisés pour la tolérance aux fautes, la situation est différente car le protocole de sauvegarde de points de reprise doit être appliqué tout au long de l'exécution de l'application. La solution par défaut de XtreemGCP est de sauvegarder périodiquement un point de reprise coordonné du travail, en utilisant son protocole. C'est alors le checkpointeur de travaux qui lance les ordres de sauvegarde d'un point de reprise. La fréquence de sauvegarde peut être choisie par l'utilisateur lors de la soumission du travail. S'il ne le fait pas, cette fréquence est choisie et adaptée par le checkpointeur de travaux en fonction de différentes informations qu'il peut obtenir comme le temps nécessaire à la sauvegarde d'un point de reprise et la fréquence des défaillances dans la grille.

Nous montrons dans le chapitre 5 que les protocoles de sauvegarde de points de reprise coordonnés ne sont pas toujours les mieux adaptés pour la tolérance aux fautes des applications distribuées. Comme nous l'avons précisé, la mise en œuvre d'autres protocoles au sein du service est envisagée. Cependant l'intégration, par exemple, de protocoles à enregistrement de messages est plus complexe que les protocoles fondés sur les points de reprise coordonnés. En effet, il n'existe pas alors réellement d'image consistante de l'application. L'image des processus est gérée d'un côté et les messages enregistrés de l'autre. De plus, certains protocoles nécessitent de maintenir plusieurs points de reprise d'un même processus. C'est pourquoi, nous estimons qu'il est alors plus pertinent de s'appuyer sur les mécanismes fournis par les bibliothèques de communication pour appliquer le protocole

de tolérance aux fautes. Le protocole étant intégré à la bibliothèque, il peut gérer par lui-même les données qu'il génère.

#### 3.2.4.4 Représentation d'un point de reprise d'un travail

Nous présentons ici le cas d'un point de reprise coordonné généré par notre protocole intégré par défaut au service. De nombreuses informations doivent être contenues dans le point de reprise coordonné pour être capable de redémarrer le travail. Notre point de reprise est composé d'un fichier de méta-données contenant différentes informations sur le travail et des fichiers de données générés par les checkpointeurs d'éléments.

Le fichier de méta-données est créé par le checkpointeur de travaux quand il sait que le point de reprise sauvegardé est valide, c'est-à-dire qu'il a reçu un acquittement de tous les checkpointeurs d'éléments. Ce fichier de méta-données contient tout d'abord la liste des fichiers générés par les checkpointeurs d'éléments. Chaque checkpointeur d'élément associe au fichier de données qu'il génère, un fichier décrivant les besoins associés à l'élément. Le checkpointeur de travaux a ensuite la charge de regrouper ces informations dans le fichier de méta-données. Ces informations contiennent notamment le checkpointeur de processus qui a été utilisé pour sauvegarder l'élément. Elles peuvent contenir aussi des informations sur les ressources utilisées par l'élément. Par exemple, si un élément exige d'être redémarré avec le même *PID*, cette information doit être enregistrée dans le fichier de méta-données. Enfin sont incluse dans le fichier de méta-données, les informations sur les besoins du travail, telles que décrites par l'utilisateur dans le fichier JSDL lors de la soumission du travail.

#### 3.2.4.5 Sélection de ressources pour le redémarrage automatique d'un travail

Après la défaillance d'un travail, il faut le redémarrer. Si cette défaillance a pour origine une défaillance matérielle, on peut alors supposer que le nœud défaillant ne sera pas disponible lors du redémarrage. Il faut donc trouver un nouveau nœud pour remplacer le nœud défaillant. C'est le checkpointeur de travaux qui à la charge de contacter le service de gestion de ressources pour obtenir les ressources nécessaires au remplacement du nœud défaillant. Lors de sa requête, il inclut les informations disponibles dans le fichier de méta-données, notamment le checkpointeur de processus utilisé lors de la sauvegarde, pour obtenir un nœud capable de redémarrer l'élément.

La découverte et l'allocation de ressources dans une grille est une tâche complexe. Obtenir de nouveaux nœuds pour remplacer les nœuds défaillants peut prendre du temps et diminuer l'efficacité du protocole de recouvrement. C'est pourquoi, il est possible d'allouer quelques nœuds supplémentaires à un travail pour, en cas de défaillance, avoir tout de suite des nœuds disponibles pour redémarrer les éléments défaillants. Cette décision dépend cependant des politiques appliquées au sein de l'organisation virtuelle et de la qualité de service requise pour le travail.

### 3.2.5 Gestion des données

La sauvegarde de points de reprise d'applications distribuées peut générer de très grandes quantités de données à sauvegarder. La bonne gestion de ces données est donc un point important pour XtreemGCP. Cependant cette gestion dépend du protocole de recouvrement arrière utilisé. Par exemple, si le protocole utilisé est fondé sur la sauvegarde de points de reprise non coordonnés, plusieurs points de reprise d'un même processus doivent être conservés à cause du risque d'effet domino. Au contraire, avec un protocole

de sauvegarde de points de reprise coordonnés, seul le dernier point de reprise valide est nécessaire pour être capable de redémarrer l'application. La gestion de ces données doit donc être prise en charge au niveau où est mis en œuvre le protocole de sauvegarde des points de reprise.

XtreemGCP fournit par défaut une solution de sauvegarde de points de reprise coordonnés. Nous présentons maintenant les règles que nous utilisons pour ce protocole.

Comme la tolérance aux fautes est un service rendu aux utilisateurs, il pourrait sembler logique de demander aux utilisateurs de fournir l'espace de stockage nécessaire à la sauvegarde des points de reprise. Cependant XtreemGCP peut aussi être utilisé pour déplacer un travail sur la grille. Dans ce cas, le service de gestion des ressources de XtreemOS est à l'origine de la migration. Un point de reprise coordonné du travail peut donc être sauvegardé sans que l'utilisateur n'ait exigé de tolérance aux fautes pour son application. C'est pourquoi XtreemGCP doit avoir à disposition de l'espace de stockage pour sauvegarder un point de reprise coordonné pendant la migration ou la suspension d'un travail.

De même, pour répondre à l'objectif de transparence du point de vue des utilisateurs, XtreemGCP doit disposer de l'espace de stockage nécessaire à mettre en œuvre une politique de tolérance aux fautes fondée sur la sauvegarde de points de reprise coordonnés. Si l'utilisation de la grille est payante, ce service sera alors facturé à l'utilisateur. Dans ce cas, le checkpointeur de travaux est chargé de supprimer les données inutiles : lorsqu'un nouveau point de reprise coordonné est validé, le précédent peut être supprimé. Lorsque le travail se termine, toutes les données de point de reprise sont supprimées.

L'utilisateur peut avoir des besoins particuliers : il veut conserver plus d'un point de reprise coordonné pour faire du débogage ; il ne veut pas que les données de points de reprise soient détruites à la fin de l'exécution du travail ; il veut que soit appliqué un autre protocole de tolérance aux fautes plus efficace pour son application. Dans ce cas, il doit alors fournir l'espace de stockage nécessaire au stockage des données, et a la responsabilité de supprimer les données de points de reprise de son application.

### 3.2.6 Interface pour les utilisateurs de XtreemGCP

L'utilisateur du service XtreemGCP peut être l'utilisateur de la grille qui soumet une application à exécuter ou une application de grille manipulant des travaux.

#### 3.2.6.1 L'utilisateur de la grille

Lors de la soumission d'un travail, nous proposons que l'utilisateur puisse fournir en plus du fichier JSDL décrivant son travail, un fichier décrivant les besoins en tolérance aux fautes de ce travail. Si l'utilisateur exige que des mécanismes de tolérance aux fautes soient appliqués à son travail, il peut alors fournir un ensemble d'informations pour décrire ses besoins.

- *Fréquence des points de reprise* : Si l'utilisateur choisit la stratégie par défaut de sauvegarde de points de reprise coordonnés, il peut spécifier la fréquence de sauvegarde des points de reprise.
- *Gestion des points de reprise* : L'utilisateur peut demander à conserver les $n$ derniers points de reprise ou que les points de reprise ne soient pas détruits à la fin du travail. Dans ce cas, il doit spécifier l'emplacement où ces points de reprise doivent être stockés.
- *Sélection du protocole* : L'utilisateur peut avertir qu'il veut utiliser un protocole de recouvrement arrière fourni par une bibliothèque de communication.

– *Sélection du checkpointeur de processus* : L'utilisateur peut définir directement quel checkpointeur de processus il veut utiliser, par exemple BLCR. Il peut aussi aider le service à choisir le bon checkpointeur de processus en donnant des indications sur les caractéristiques de son travail. Il peut par exemple spécifier si son travail est composé de processus multi-threads, si les identifiants de processus doivent être sauvegardés et restaurés, *etc*

**Gestion des fichiers ouverts par le travail** Lors de la sauvegarde d'un point de reprise d'un travail, la gestion des fichiers ouverts par ce travail est un problème complexe. En effet, le travail peut utiliser ou générer des fichiers de données de très grande taille. Dans ce cas, il serait très coûteux de sauvegarder ces fichiers dans le point de reprise. D'un autre côté, un de ces fichiers peut être indispensable au redémarrage du travail. Si un travail est suspendu pour être redémarré plus tard, il est alors impossible d'assurer que ce fichier sera toujours disponible au redémarrage. C'est pourquoi nous offrons aux utilisateurs la possibilité de donner une liste des fichiers à sauvegarder avec les points de reprise. Une fonction de rappel est alors chargée de la sauvegarde des fichiers associés à un élément.

#### 3.2.6.2 Application de grille

Nous avons vu dans le paragraphe 3.1.5.3, que les travaux menés par l'OGF sur le recouvrement arrière dans les grilles, ont menés à l'introduction de quatre fonctions dans l'interface SAGA : *suspend()*, *resume()*, *checkpoint()* et *migrate()*. Nous mettons en œuvre ces quatre appels dans XtreemOS pour que des applications de grille puissent manipuler des travaux. L'appel à l'un de ces appels transmet simplement la commande au checkpointeur de travaux concerné.

## 3.3 Synthèse

Dans ce chapitre nous avons présenté XtreemGCP, un service de recouvrement arrière pour la grille. Intégré au système de grille XtreemOS, il permet d'assurer la bonne terminaison des travaux exécutés sur la grille en dépit des défaillances. De plus, il offre la possibilité de suspendre ou de déplacer un travail pour optimiser la gestion des ressources dans la grille. Fondé sur une architecture complètement distribuée assurant son passage à l'échelle, XtreemGCP est capable de gérer des travaux composés de plusieurs éléments distribués sur différents nœuds de la grille.

Les travaux sur les outils de sauvegarde de processus sont nombreux et la recherche dans ce domaine est active. C'est pourquoi nous proposons une interface générique simple pour les outils de sauvegarde de processus, permettant de découpler notre service d'un outil particulier. De même, des bibliothèques de communication telles que Open MPI, ayant une approche modulaire pour l'intégration de ces outils de sauvegarde de processus, peuvent tirer partie de cette interface générique pour utiliser les outils de sauvegarde de processus disponibles sur les nœuds de la grille XtreemOS. Aujourd'hui BLCR et OpenVZ ont été intégrés au service par les partenaires du projet XtreemOS. D'autres solutions pourront l'être dans le futur.

Pour une utilisation simple du service, nous proposons une stratégie par défaut de sauvegarde de points de reprise coordonnés pour le service. Les mécanismes, mis en œuvre au niveau de l'interface *socket* pour les communications, et exploitant les outils de sauvegarde de processus au niveau système, permettent de fournir de la tolérance aux fautes sans

modification de l'application de l'utilisateur, pour des applications à échange de messages. Ils offrent une brique de base pour la mise en œuvre de différents protocoles de recouvrement arrière. Ainsi un protocole de sauvegarde de points de reprise non coordonné a été intégrer au service [68].

Si l'utilisateur a des exigences particulières, par exemple en terme de performances, auxquelles les stratégies de base de sauvegarde de points de reprise ne peut répondre, le service peut alors exploiter les solutions fournies par des bibliothèques de communication telles que Open MPI ou MPICH-V. Nous présentons dans le chapitre 5, une solution de recouvrement arrière fondée sur l'enregistrement de messages et visant des applications distribuées de grande taille. Cette solution, intégrée à Open MPI, pourrait être utilisée dans ce contexte.

Dans le cas d'un utilisateur connaissant les problématiques de sauvegarde de points de reprise dans la grille, ou connaissant au moins les caractéristiques de son application, les informations supplémentaires qu'il peut fournir au service peuvent permettre de prendre de meilleures décisions. C'est pourquoi nous offrons aux utilisateurs la possibilité de décrire leurs besoins lors de la soumission d'un travail.

Enfin XtreemGCP s'appuie sur les autres services de XtreemOS pour assurer une gestion transparente de la tolérance aux fautes. Il se sert du service de gestion de ressources de XtreemOS pour trouver, avec l'aide des informations stockées par le service dans le fichier de méta-données associé au point de reprise, des nœuds capables de redémarrer un travail défaillant et ainsi peut redémarrer automatiquement ce travail. XtreemGCP s'appuie aussi sur XtreemFS pour sauvegarder les données de point de reprise de façon stable et les rendre accessibles depuis tous les nœuds de la grille. La gestion des données générées par les mécanismes de recouvrement arrière étant un point critique, nous proposons un ensemble de règles de base pour traiter ces données.

# Chapitre 4

# Semias, cadre pour la réalisation de services de grille hautement disponibles et auto-réparants

Dans ce chapitre, nous présentons Semias, cadre fournissant à des services de grilles les propriétés de haute disponibilité et d'auto-réparation.

Semias est fondé sur l'utilisation combinée d'un réseau pair-à-pair structuré et de techniques de duplication active. Grâce à la combinaison de ces deux techniques, l'effort de programmation nécessaire pour intégrer un service existant à Semias est minime. De plus, la duplication est rendue complètement transparente pour les clients d'un service dupliqué qui ne voient qu'un service hautement disponible. Semias offre donc une brique de base pour la conception de systèmes de grille réellement adaptés aux grilles de grande taille et dynamiques.

Ce travail a été validé dans le contexte du système de grille Vigne [159]. Semias a été utilisé pour rendre le service de gestion d'applications de Vigne hautement disponible. Les évaluations menées sur la plate-forme d'expérimentation Grid'5000 mettent en évidence les capacités de Semias à assurer la disponibilité de services déployés dans un environnement dynamique.

Les travaux présentés dans ce chapitre sont le fruit de collaborations avec Rajib Nath, étudiant en Master à *University of Tennessee* (État-Unis), Sébastien Gillot, étudiant en Master à l'Université de Rennes 1, et Stefania Costache, étudiante en Master à *Politehnica University of Bucarest* (Roumanie).

L'organisation de ce chapitre est la suivante. Dans le paragraphe 4.1, nous décrivons le contexte de ce travail, ses objectifs détaillés et les problèmes ouverts. Le paragraphe 4.2 présente la pile logicielle de Semias. Nous y mettons en avant nos solutions pour une bonne gestion de la taille et de la volatilité des ressources des grilles de calcul. L'évaluation de Semias dans le cadre du système de grille Vigne est décrite dans le paragraphe 4.3. Enfin nous dressons un bilan de ces travaux dans le paragraphe 4.4.

## 4.1 Problématique

Dans ce paragraphe, nous commençons par détailler le contexte de ce travail et notamment le système de grille Vigne. Nous présentons ensuite les objectifs auxquels doit répondre Semias. Nous mettons en évidence l'intérêt de la combinaison de techniques pair-à-pair et de duplication active, ainsi que les problèmes ouverts. Enfin nous présentons une

étude de l'état de l'art. Elle met en évidence l'originalité de ces travaux dans le domaine des grilles de calcul, et décrit les travaux sur la haute disponibilité ayant servi de base pour la conception de Semias.

### 4.1.1 Contexte : l'intergiciel de grille Vigne

Vigne est un système de grille adapté aux grilles hétérogènes développé au sein de l'équipe-projet PARIS à l'INRIA Rennes Bretagne Atlantique. Il vise essentiellement les grilles composées de stations de travail et de grappes de calcul. L'architecture de Vigne est décrite par la figure 4.1.

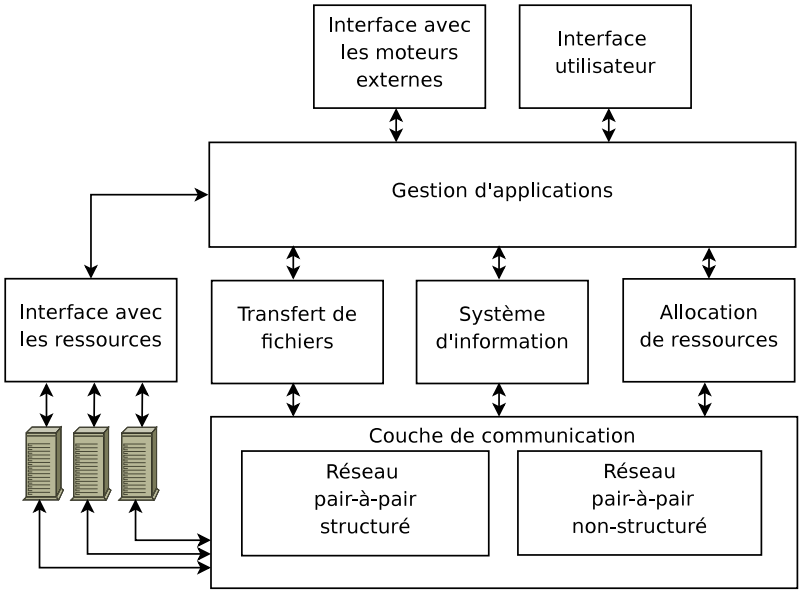



Fig. 4.1 – Architecture de Vigne

Le service de gestion d'applications est le service principal de Vigne. C'est lui qui a la charge de l'application lors de son cycle de vie sur la grille, de sa soumission par l'utilisateur jusqu'à l'obtention des résultats :

- Vigne fournit une interface aux utilisateurs pour soumettre une application à la grille, obtenir des informations sur les applications en cours d'exécution, *etc.* Cette interface est accessible depuis n'importe quel nœud de la grille. Lors de la soumission, l'utilisateur décrit son application et les besoins associés à celle-ci à l'aide d'un fichier JSDL [12]. Une interface pour des moteurs externes, comme un moteur de workflow [165], est aussi fournie. Elle permet notamment de gérer des dépendances entre des tâches.
- Une fois l'application soumise, le gestionnaire d'application doit trouver des ressources pour l'exécuter. Pour cela, il s'appuie tout d'abord sur le système d'information [101]. Celui-ci fournit une liste de ressources répondant aux critères spécifiés par l'utilisateur. Le service d'allocation de ressources est ensuite chargé de sélectionner les ressources les plus appropriées parmi cette liste pour optimiser les performances de l'application [102].
- Avant d'exécuter l'application sur les ressources sélectionnées, le gestionnaire d'applications utilise le service de transfert de fichiers pour transférer les exécutables et les fichiers de données associées à cette application vers ces ressources. Ce service

est à nouveau utilisé pour récupérer les fichiers de résultat à la fin de l'exécution.
- Enfin l'interface avec les ressources permet la soumission d'application sur des ressources hétérogènes. Cette interface permet notamment de soumettre une application sur des nœuds exploités avec Linux ou sur des nœuds exploités avec un gestionnaire de traitement par lot tel que Torque [190].
- Durant l'exécution de l'application, le gestionnaire d'application est chargé de superviser les ressources sur lesquelles l'application s'exécute pour détecter les défaillances [160]. En cas de défaillance, le gestionnaire d'application peut automatiquement relancer l'application sur d'autres ressources.

Ainsi Vigne permet aux utilisateurs de profiter simplement des ressources de la grille pour exécuter leurs applications.

Le service de gestion d'application est conçu sur le même principe que XtreemGCP : un nouveau gestionnaire d'application est créé pour chaque application. Les gestionnaires d'application sont répartis sur l'ensemble des nœuds de la grille en utilisant un réseau pair à pair structuré. Nous décrivons les principes des réseaux pair à pair structurés dans la section 4.1.4.1.

### 4.1.2 Objectifs

Pour rendre des services de grille tels que le service de gestion d'applications de Vigne ou encore XtreemGCP hautement disponible et auto-réparant, nous proposons la pile logicielle nommée Semias. Voici le cahier des charges auquel doit répondre Semias pour être adapté aux grilles de calcul.

**Performances** La haute disponibilité se fonde sur de la duplication. Dupliquer un service, c'est-à-dire créer plusieurs instances de ce service qui doivent être synchronisées, augmente la quantité de ressources nécessaire à son exécution et diminue ses performances. Cependant fournir des services hautement disponibles ne doit pas rendre ces services inutilisables. Le surcoût induit par Semias sur la performance des services, notamment en terme de temps de réponse, doit donc être limité.

**Passage à l'échelle** Des services de grille tels que XtreemGCP ou le service de gestion d'applications de Vigne sont divisés en de multiples entités pour améliorer leur tolérance aux fautes et surtout pour passer à l'échelle. Chaque entité doit alors être rendu hautement disponible. Semias doit donc passer à l'échelle pour gérer ce grand nombre d'entités.

**Équilibrage de charge** Dupliquer de nombreux services peut induire une charge importante. Semias doit être capable d'exploiter tous les nœuds de la grille pour distribuer la charge des duplicatas de services à exécuter.

**Simplicité d'utilisation** Pour être attractive, l'utilisation de Semias ne doit requérir que très peu de modifications des services existants pour les rendre hautement disponibles. Cela signifie que l'effort pour intégrer un service à Semias doit être minimal et que la duplication des services doit être transparente pour les clients pour éviter d'avoir à modifier ceux-ci.

### 4.1.3 Modèle considéré

Notre étude porte sur la duplication de services de grille, services ayant un état, pour les rendre hautement disponibles. Le modèle de fautes que nous considérons est celui que nous avons décrit dans le chapitre 1. Nous considérons le modèle de partition primaire [158], c'est-à-dire qu'une seule vue d'un groupe de duplicatas peut être valide à un instant donné.

### 4.1.4 Vers des services de grille hautement disponibles et auto-réparants

Semias est fondé sur l'utilisation combinée de deux technologies bien connues que sont les réseaux pair à pair structurés et la duplication active. Dans ce paragraphe, nous présentons ces deux technologies et les propriétés qu'elles apportent. Nous décrivons ensuite les principes de fonctionnement de Semias. Enfin nous mettons en évidence les problèmes ouverts par la combinaison de ces deux techniques.

#### 4.1.4.1 Les réseaux pair à pair structurés

Un réseau pair-à-pair structuré, encore appelé réseau logique structuré, est une abstraction du réseau physique offrant des primitives de routage des messages fondées sur des clés. Plusieurs réseaux logiques structurés existent tels que CAN [154], Chord [182], Pastry [163] ou Tapestry [206]. Ces réseaux fournissent des mécanismes de routage extensibles et tolérant aux fautes. Ils permettent de mettre en œuvre une table de hachage distribuée. Pour illustrer le fonctionnement d'un réseau logique structuré, nous prenons l'exemple de Pastry, qui est le réseau que nous avons utilisé pour nos travaux.

Dans Pastry, les nœuds du réseau logique sont organisés en anneau. À chaque nœud du réseau est associé un identifiant unique choisi aléatoirement, que nous représentons sous forme hexadécimale. Les nœuds sont placés sur l'anneau dans le sens horaire. À chaque objet inséré dans la table de hachage distribuée est associé une clé unique. Le nœud dans le réseau logique dont l'identifiant est le plus proche de la clé d'un objet est responsable de cet objet. L'accès à cet objet se fait par ce nœud. Chaque nœud dans Pastry tient à jour une table de routage et une liste de voisins pour pouvoir router les messages à partir d'une clé en suivant l'anneau logique. Cette liste de voisins est composée de $2 * L$ voisins, les $L$ voisins les plus proches du nœuds de chaque côté de celui-ci. Un message est routé vers le nœud dont l'identifiant est le plus proche de la clé en $O(logN)$ étapes, N étant le nombre de nœuds dans le réseau. Chaque nœud du réseau logique surveille ses voisins pour détecter les défaillances et des chemins redondant sont utilisés pour arriver à router les messages en dépit des reconfigurations.

La figure 4.2 illustre le routage des messages fondé sur des clés dans Pastry. Comme le montre la figure, le routage fondé sur des clés permet d'accéder à un objet sans connaître sa position réelle dans le réseau.

Les objets que nous voulons placer dans notre table de hachage distribuée sont des services de grille. Un choix aléatoire des clés associées aux services assure une répartition uniforme des services sur l'ensemble des nœuds du système.

#### 4.1.4.2 La duplication active de services

Comme nous l'avons vu dans la section 1.2.4.3, les techniques de tolérances aux fautes les mieux adaptées pour des entités telles que des services ayant des interactions avec le monde extérieur sont les techniques de duplication. Les deux principales techniques de duplication existantes sont la duplication active et la duplication passive [201].

Quand un service est dupliqué passivement, seul le duplicata principal traite les requêtes. Les duplicatas passifs sont mis à jour par des messages de mise à jour générés par le duplicata principal. L'avantage principal ici est que la requête n'est traitée qu'une seule fois. Cependant il est nécessaire de mettre en œuvre les mécanismes pour générer et traiter les messages de mise à jour.

Quand un service est dupliqué activement, tous les duplicatas du service traitent les requêtes. Le service dupliqué doit avoir un comportement déterministe. Assurer que les

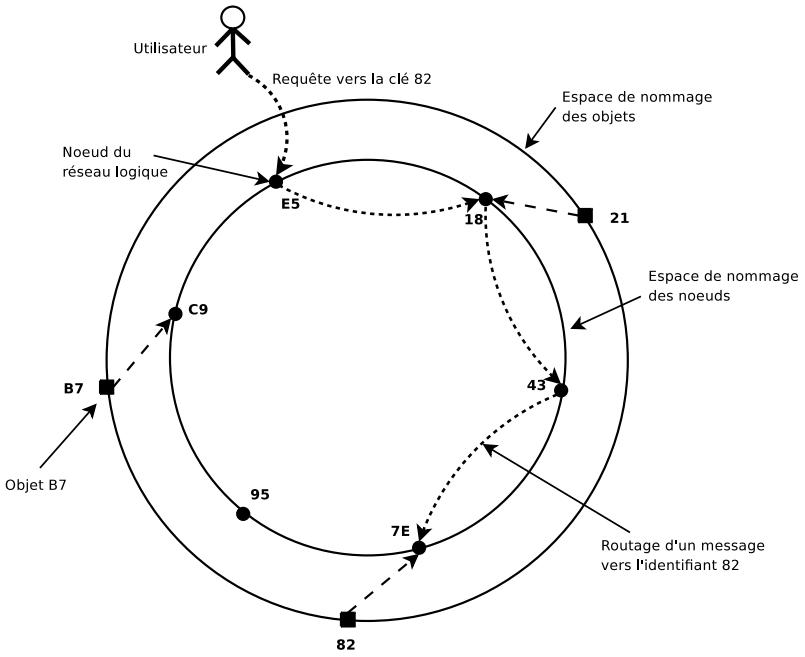



FIG. 4.2 – Routage d'un message dans Pastry

duplicatas délivrent le même ensemble de requêtes dans le même ordre est alors suffisant pour assurer la cohérence de ceux-ci. L'utilisation d'une primitive de communication de groupe de type diffusion atomique [41] offre ces propriétés. Le coût de la duplication active est supérieur à celui de la duplication passive car toutes les requêtes sont traitées par tous les duplicatas. Cependant, il n'est alors pas nécessaire de modifier le service pour le dupliquer.

Les techniques de duplication actives peuvent avoir de meilleures performances quand il n'y a pas de défaillances car un client peut prendre en compte la première réponse lui arrivant d'un des duplicatas, alors que pour un service dupliqué passivement, le temps de réponse dépend uniquement des performances du duplicata principal. De plus, elles ont de meilleures performances en cas de reconfiguration. Avec la duplication active, si un duplicata actif est défaillant, les autres duplicatas actifs peuvent continuer à traiter les requêtes. Avec la duplication passive, si le duplicata principal est défaillant, une reconfiguration est nécessaire pour élire un nouveau duplicata principal avant de traiter de nouveaux messages.

D'autres stratégies de duplication ont été proposées. La duplication semi-passive [56] fonctionne selon le même principe que la réplication passive, c'est-à-dire que chaque requête est traitée par un seul duplicata, mais assure un temps de réponse bien meilleur en cas de défaillance en évitant la reconfiguration du groupe de duplicatas à chaque suspicion de défaillance. La duplication semi-active [43] fonctionne comme la duplication active mais permet de prendre en charge les comportements non déterministes des services : quand des parties associées au traitement d'une requête sont non déterministes, un seul duplicata exécute ces traitements et met à jour les autres duplicatas comme cela est fait pour la duplication passive.

L'analyse des services de Vigne nous a montré que leur comportement était déterministe. C'est pourquoi les meilleures performances et la plus grande transparence offertes par les solutions de duplication active nous ont amené à opter pour ce type d'ap-

proche.

#### 4.1.4.3 Duplication active dans un réseau pair à pair structuré

Dans ce paragraphe nous présentons l'idée associée à l'utilisation de techniques de duplication active dans un réseau pair à pair structuré. Il ne s'agit ici que d'une présentation d'introduction. Le détail du placement des duplicatas d'un service dans le réseau logique est décrit dans la section 4.2.3.

Dans Semias, nous plaçons les $n$ duplicatas d'un service sur les $n$ nœuds du réseau logique structuré dont les identifiants sont les plus proches de la clé du service. Ceci est illustré par la figure 4.3, dans laquelle nous avons placé deux services dupliqués, ayant pour clé 58 et $AF$. Dans cet exemple, le degré de duplication est de trois.

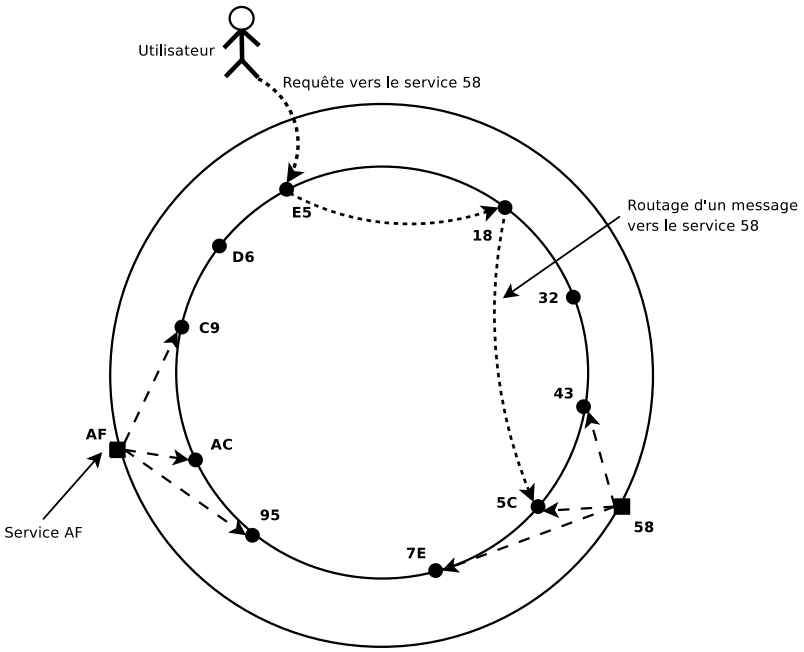



Fig. 4.3 – Duplication active de services au dessus de Pastry

Dans Pastry, un message est routé vers le nœud dont l'identifiant est le plus proche de la clé du message. En plaçant les duplicatas d'un service sur les nœuds les plus proches[1] de la clé du service, nous assurons que les requêtes envoyées à un service atteignent toujours un nœud hébergeant un duplicata du service : si le nœud dont l'identifiant est le plus proche de la clé est défaillant, le nouveau nœud le plus proche de la clé héberge lui aussi un duplicata du service. Dans l'exemple de la figure 4.3, si le nœud $5C$ est défaillant, les messages adressés au service 58 vont être routés vers le nœud 43 qui héberge lui aussi un duplicata du service.

L'utilisation d'un réseau pair à pair structuré offre une solution de routage dans la grille passant à l'échelle et tolérante aux fautes. De plus, distribuer les services sur les nœuds du réseau logique permet de résoudre le problème du positionnement des duplicatas d'un service dans la grille et de l'équilibrage de charge. Enfin la duplication active assure la haute disponibilité des services.

[1]Dans ce chapitre, la notion de proximité fait toujours référence à la proximité des identifiants.

#### 4.1.4.4 Problèmes ouverts

Le problème ouvert par la mise en place de duplication active au-dessus d'un réseau pair à pair structuré est celui de la gestion de l'impact des reconfigurations dans le réseau logique sur la composition des groupes de duplicatas. Deux cas sont à traiter : celui du retrait de nœuds défaillants du réseau logique et celui de l'ajout de nœuds dans le réseau logique. Nous ne prévoyons pas de mécanismes spécifiques pour les retraits volontaires de nœuds. Nous les traitons donc comme des défaillances.

**Retrait de nœuds** Après la défaillance d'un nœud hébergeant un duplicata d'un service, il faut créer un nouveau duplicata de ce service pour maintenir son degré de duplication et donc son niveau de disponibilité. Pour offrir la propriété d'auto-réparation, Semias doit être capable de gérer ce type de reconfiguration automatiquement.

**Ajout de nœuds** À tout moment, de nouveaux nœuds peuvent être intégrés à la grille et donc au réseau logique. Le nombre de nouveaux nœuds peut être important, par exemple quand les ressources d'un nouveau site sont connectées à la grille. Le système peut alors se trouver dans un état tel que décrit par la figure 4.4. Sur cette figure, nous reprenons l'exemple du service 58 de la figure 4.3. Avec l'intégration de nouveaux nœuds au réseau logique, les duplicatas du service 58 ne se trouvent plus sur les nœuds les plus proches de la clé. Dans cette configuration, si un message est envoyé avec la clé 58, il sera reçu par le nœud 59 qui n'héberge pas de duplicatas du service. C'est pourquoi, lors de l'ajout de nouveaux nœuds, il est aussi nécessaire de reconfigurer les groupes de duplicatas déjà présents dans le système, pour assurer que les duplicatas d'un service soient sur les nœuds les plus proches de la clé du service.

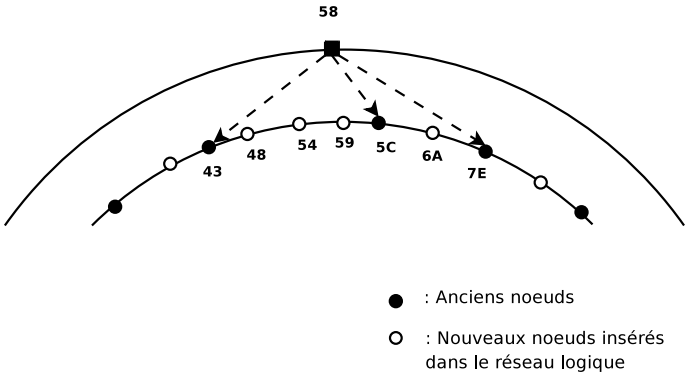



Fig. 4.4 – Configuration possible du réseau logique après l'ajout de plusieurs nœuds

**Reconfigurations et duplication active** Reconfigurer des groupes de duplicatas de services dupliqués activement peut être coûteux. En effet, créer un nouveau duplicata d'un service implique de transférer l'état du service depuis un duplicata existant vers le nœud hébergeant le nouveau duplicata pour permettre l'initialisation de celui-ci. Ce transfert d'état peut être coûteux selon la taille de l'état.

Comme l'illustre notamment la figure 4.4, dans notre situation le nombre de reconfigurations à appliquer aux groupes de duplicatas pour s'adapter aux changements dans le réseau pair à pair peut être très important. Une manière simple de gérer les changements

dans le réseau logique serait de répercuter chaque changement aux groupes de duplicatas concernés. Cependant dans certaines situations, cette solution entraînerait des reconfigurations inutiles. Prenons l'exemple de la figure 4.5 où un nouveau nœud, *54*, vient d'être intégré au réseau logique. La reconfiguration qui doit alors avoir lieu est de *déplacer* un duplicata du service *58* du nœud *43* vers ce nouveau nœud. Ceci implique d'ajouter le nouveau nœud dans le groupe, de transférer l'état du service vers ce nouveau nœud, et de supprimer le nœud *43* du groupe. Cependant, si pendant cette reconfiguration d'autres modifications interviennent dans le réseau pair à pair, cette première reconfiguration peut devenir inutile : si le nœud *5C* est défaillant, le nœud *43* devra être réintégré au groupe pour conserver le degré de duplication ; si de nouveaux nœuds plus proches de la clé *58* que *54* apparaissent dans le réseau logique, le nœud *54* devra à son tour être retiré du groupe. C'est pourquoi nous voulons proposer une solution permettant de limiter le nombre de reconfigurations de groupes de duplicatas tout en assurent le bon fonctionnement et la disponibilité des services dupliqués..

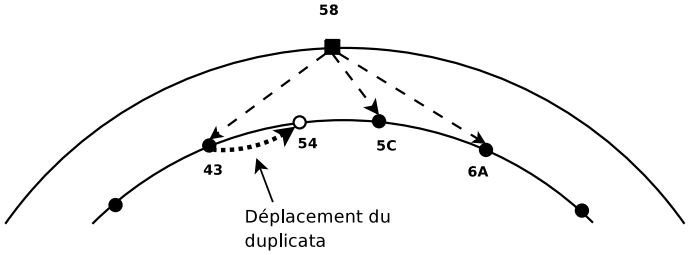



Fig. 4.5 – Reconfiguration d'un groupe après ajout d'un nouveau nœud

En résumé, notre objectif est d'avoir une solution capable de supporter des communications de groupe dynamiques pour pouvoir appliquer des reconfigurations aux groupes de duplicatas. De plus, nous voulons une solution capable de reconfigurer les groupes de duplicatas en fonction des reconfigurations du réseau logique. Enfin nous voulons mettre en place une politique de gestion des groupes de duplicata permettant de limiter le nombre de reconfigurations.

### 4.1.5 Travaux apparentés

Nous traitons les travaux apparentés en deux parties. Dans un premier temps, nous présentons les travaux relatifs à la haute disponibilité dans les grilles de calcul. Dans un deuxième temps, nous étudions les travaux sur les problèmes de duplication active dans les environnements dynamiques et plus particulièrement dans les réseaux pair-à-pair structurés.

#### 4.1.5.1 Haute disponibilité des systèmes de grilles

Nous commençons par présenter des systèmes de grille qui pourraient tirer partie d'une solution telle que Semias pour fournir à leurs utilisateurs des services hautement disponibles et auto-réparants.

Globus [74] repose sur une hiérarchie statique de services définie par les administrateurs. Ceci implique que certains nœuds ont un rôle particulier et que la défaillance de ces nœuds peut entraîner la défaillance du système, ou du moins que traiter la défaillance de ces nœuds requiert une intervention humaine. Nous estimons que donner un rôle particulier

à certains nœuds de la grille n'est pas une bonne solution car tout nœud de la grille peut être amené à subir une défaillance. Même si les services de Globus sont conçus pour être tolérant aux défaillances, ils sont souvent centralisés [69], ce qui limite leur extensibilité. Des travaux récents [151] s'intéressent au déploiement dynamique de services Globus dans la grille, mais n'examinent pas le problème des reconfigurations pour les services déjà déployés.

Legion [118] est un système de grille complètement décentralisé et donc extensible. Legion est composé d'objets ayant des identifiants indépendants de leur position réelle dans la grille. Ainsi un objet défaillant peut être redémarré sur un autre nœud de la grille. Cependant les mécanismes de tolérance aux fautes appliqués aux objets sont fondés sur la sauvegarde de points de reprise. Le redémarrage d'un objet fautif peut donc entraîner un retour arrière.

Dans DIET [8], des agents distribués sont chargés de répertorier les ressources disponibles dans la grille pour pouvoir répondre aux requêtes des clients qui peuvent ensuite contacter directement ces ressources. Comme dans XtremWeb, une mise en œuvre simple consiste à avoir un seul agent centralisé. Une solution alternative a été proposée [39], fondée sur une hiérarchie d'agents distribués interconnectés par un réseau pair à pair. Cette solution est plus extensible et offre une meilleure tolérance aux fautes qu'une solution centralisée car la défaillance d'un agent n'entraîne pas l'arrêt du système. Cependant, la défaillance d'un agent implique la perte des données stockées par celui-ci.

Nous présentons maintenant les solutions choisies par d'autres systèmes de grille pour traiter le problème de la haute disponibilité.

Vishwa [155] est un système de grille qui se fonde sur la duplication des données associées aux services de grille dans une table de hachage distribuée pour assurer la disponibilité et l'extensibilité du système. Cependant, nous n'avons pas connaissance de travaux décrivant les moyens utilisés pour assurer la cohérence des services dupliqués ni pour gérer les cas de reconfiguration. Zorilla [62] est lui aussi fondé sur une table de hachage distribuée, ce qui lui permet d'être extensible et complètement distribué. Cependant les cas de reconfiguration ne sont pas non plus traités dans Zorilla.

XtreemOS propose une solution alternative pour la haute disponibilité de services de grille [149] fondée sur une solution de duplication active combinée avec des techniques de type IPv6 mobile [188] pour rendre la duplication des services transparente pour les clients. Cependant cette solution ne traite pas du problème du positionnement des services dupliqués et donc de l'équilibrage de charge.

Dans Migol, une solution de duplication active est proposée pour le système d'information [124]. L'article montre qu'il est possible d'utiliser des techniques de duplication active dans les grilles et propose un nouveau protocole de diffusion atomique dans ce but. Cependant les cas de reconfiguration ne sont pas évalués et le problème du positionnement des duplicatas n'est pas traité. Les travaux se concentrent sur les problèmes liés à la sécurité.

Les auteurs de [205] utilisent de la duplication passive pour des services de grille car ils veulent être capables de dupliquer des services au comportement non déterministe. Les défaillances ne sont pas transparentes pour les clients qui doivent explicitement se reconnecter en cas de défaillance du duplicata principal. De plus l'étude est limitée à des duplicatas s'exécutant sur la même grappe de calcul.

XtremWeb [38] est fondé sur le modèle maître-travailleurs. Un maître, aussi appelé coordinateur, prend en charge les requêtes des utilisateurs et assigne les tâches aux travailleurs. Le coordinateur est donc le composant le plus important du système. Cependant, ce coordinateur est mis en œuvre de façon centralisée, ce qui en fait un point unique

de vulnérabilité aux défaillances et peut limiter l'extensibilité du système. Une solution fondée sur la duplication passive [60] a été proposée pour rendre le coordinateur hautement disponible.

Aucun de ces travaux ne propose de solution pour l'ensemble des problèmes que nous nous proposons de traiter.

#### 4.1.5.2 Duplication active dans un environnement dynamique

La duplication active se fonde sur la diffusion atomique des messages à un groupe de duplicatas. Le modèle de groupe dynamique permet à la composition d'un groupe de changer au cours du temps. Ainsi, un nouveau membre peut être ajouté pour remplacer un membre défaillant [128]. C'est donc ce type de groupe qui nous intéresse dans le cadre de Semias.

**Définition 4.1 (Groupe dynamique)** *Groupe de processus dont la composition peut changer au cours du temps.*

Les changements de composition dans un groupe dynamique sont appelés des changements de vues et sont pris en charge par un module de composition de groupes (*Group Membership*). La diffusion atomique dans un groupe de processus est alors définie par cinq propriétés [167]. Nous nous intéressons ici à la diffusion atomique *uniforme*, qui contraint le comportement de tous les processus du groupe, même les fautifs, pour assurer que toute réponse à une requête reçue par un client soit valide. Dans ce paragraphe, *diffuser* est l'envoi *atomique* d'un message à tous les membres d'un groupe ; *délivrer* est la réception par un processus du groupe d'un message *diffusé* dans le groupe.

**Propriété 4.1 (Validité)** *Si un processus correct diffuse un message dans le groupe, il délivre ultimement ce message.*

**Propriété 4.2 (Accord uniforme)** *Si un processus $p_i$ délivre un message $m$ dans une vue $v$, alors tous les processus corrects de la vue $v$ délivrent ultimement $m$.*

**Propriété 4.3 (Intégrité uniforme)** *Les processus délivrent un message $m$ au plus une fois, et ne le délivrent que si le message a été préalablement diffusé.*

**Propriété 4.4 (Livraison dans la même vue)** *Si deux processus délivrent un message, ils le font dans la même vue.*

**Propriété 4.5 (Ordre total uniforme)** *Si un processus d'une vue $v$ délivre un message $m$ avant de délivrer un message $m'$ dans cette vue, alors tous les processus de la vue $v$ ne délivre $m'$ qu'après avoir délivré $m$.*

La propriété de *livraison dans le même vue* requiert que les changements de vue soient totalement ordonnés avec la livraison des messages applicatifs. Cette propriété est aussi appelée *view synchrony* [22].

Il a été démontré que le problème de la diffusion atomique et le problème du consensus sont équivalents dans un système asynchrone où des processus peuvent subir des défaillances [41]. Le problème peut donc être résolu en utilisant un détecteur de défaillances non fiable.

Defago et al. [57] fournissent un état de l'art détaillé de l'ensemble des systèmes de communications de groupe existants offrant de la diffusion atomique. Deux principales architectures existent.

**Diffusion atomique fondée sur le module de composition de groupes** Cette solution, illustrée par la figure 4.6, est utilisée par la plupart des systèmes de communication de groupe existants, tels que ISIS [21], Phoenix [202] ou Ensemble [86]. Le module de composition de groupe est chargé d'ordonner les changements de vues et les messages destinés aux membres du groupe. Il est aussi chargé de donner une vue cohérente des membres d'un groupe en simulant un détecteur de défaillances parfait [41] au-dessus d'un détecteur de défaillances non fiable. Pour ce faire, il exclut tout membre du groupe suspecté d'être défaillant. Ainsi il assure que l'algorithme de diffusion atomique, fondé le plus souvent sur un séquenceur fixe out tournant, ne se bloque pas en cas de défaillance. Le coût d'une fausse suspicion est alors très élevé car elle implique une reconfiguration inutile du groupe.

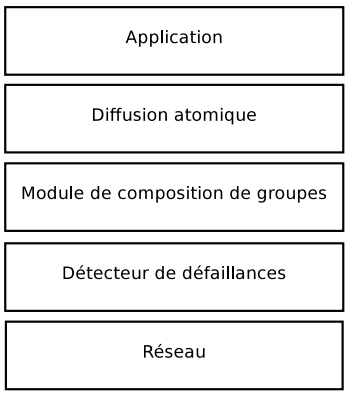



FIG. 4.6 – Diffusion atomique fondée sur un module de composition de groupes

**Module de composition de groupes fondé sur la diffusion atomique** Cette solution, illustrée par la figure 4.7, a été proposée par Mena et al. [128]. Elle semble plus naturelle car elle permet d'utiliser la diffusion atomique pour ordonner les changements de vues et les messages destinés aux membres du groupe. Cette architecture requiert d'utiliser un algorithme de diffusion atomique pouvant se fonder sur un détecteur de défaillance non fiable. Mena et al. suggèrent l'utilisation d'un algorithme de diffusion atomique fondé sur un algorithme de consensus utilisant un détecteur de défaillance non fiable de type $\diamond S$ [41]. Ainsi la suspicion d'un nœud et l'éviction de ce nœud du groupe peuvent être découplées.

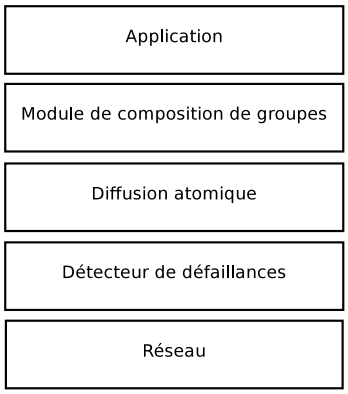



FIG. 4.7 – Module de composition de groupes fondé sur la diffusion atomique

**Définition 4.2 (Fausse suspicion)** *Une fausse suspicion a lieu quand un processus suspecte à tord un autre processus d'être défaillant.*

Dans un réseau étendu où la latence entre les nœuds peut être grande et variable, des mécanismes de détection de défaillances efficaces peuvent être difficiles à mettre en place. Fixer des délais de réponse courts peut conduire à de nombreuses fausses suspicions. Fixer des délais de réponse longs peut pénaliser les performances du système car les réelles défaillances ne sont pas alors rapidement détectées. Dans ce contexte, l'architecture proposée par Mena et al. est mieux adaptée [193] car elle permet d'utiliser des politiques de détection des défaillances différentes pour l'algorithme de consensus et pour le module de composition des groupes. Pour l'algorithme de consensus, un délai très court peut être utilisé avant de suspecter un nœud pour assurer un bon temps de réponse. Un délai beaucoup plus long peut être utilisé au niveau de la composition des groupes, pour éviter les fausses suspicions, et ainsi éviter les reconfigurations inutiles. C'est pourquoi dans la cadre de Semias, nous optons pour cette architecture.

#### 4.1.5.3 Duplication active dans les réseaux pair-à-pair structurés

À notre connaissance, le seul projet traitant le problème de la duplication active dans un réseau logique structuré est PaxonDHT [189]. Les auteurs proposent une solution pour mettre en œuvre l'algorithme de consensus Paxos [23] au-dessus de Pastry et ainsi offrir une brique de base pour la duplication active. PaxonDHT a été évalué par simulation.

Dans l'algorithme Paxos original [110], défini pour des groupes statiques (c'est-à-dire dont la composition ne change pas), l'utilisation d'un support de stockage stable est nécessaire pour assurer la sûreté de l'algorithme de consensus. Un processus doit sauvegarder sur support stable les valeurs qu'il accepte pour chaque instance du consensus. Ainsi si il subit une défaillance et redémarre, il peut récupérer les valeurs qu'il avait accepté avant la défaillance pour ne pas accepter de valeurs différentes après la défaillance pour une même instance du consensus. C'est la condition nécessaire pour qu'une unique valeur soit décidée pour chaque instance du consensus.

**Définition 4.3 (Instance de consensus)** *Une instance de consensus est une exécution de l'algorithme de consensus pour décider une valeur.*

Cependant, la mise en œuvre de Paxos au-dessus d'un réseau logique structuré implique des groupes dynamiques : le processus remplaçant un processus défaillant n'est pas nécessairement exécuté sur le même nœud. PaxonDHT suppose qu'il n'existe pas de support stable accessible depuis tous les nœuds pour assurer la sûreté de l'algorithme. Le nouveau processus ne connaît donc pas les valeurs acceptées avant la défaillance par le processus qu'il remplace. La sûreté de Paxos ne peut donc pas être assurée pour les instances de consensus auxquelles le processus défaillant a participé mais pour lesquelles une valeur n'a pas encore été décidée. Cependant PaxonDHT montre qu'avec un nombre important de processus participant au consensus, les risques de violer la condition de sûreté sont faibles. C'est pourquoi ils proposent d'avoir un nombre de duplicatas supérieur à cinq pour assurer la condition de sûreté avec une haute probabilité.

Comme nous le décrivons dans le paragraphe 4.2.2.3, Semias utilise aussi Paxos comme algorithme de consensus, mais assure simplement la sûreté de l'algorithme en se reposant sur le module de composition de groupe. En effet, Semias assure que seuls les membres de la vue courante du groupe peuvent participer aux instances de consensus. Ainsi Semias ne requiert pas un degré de duplication élevé.

Des travaux se sont intéressés au stockage d'objets dupliqués modifiables dans une table de hachage distribuée [18, 131, 133]. Ils ont donc dû résoudre le problème de validation atomique (*atomic commit*) [175] dans un réseau logique structuré. Cependant dans ce cas, ils ne sont pas contraints par la propriété de *livraison dans la même vue* associée à la diffusion atomique. La gestion des reconfigurations est donc simplifiée. Néanmoins ces projets mettent en œuvre des stratégies très basiques. Dans Etna [133] et Scalaris [131], les groupes de duplicatas sont reconfigurés à chaque changement dans le réseau logique, ce qui peut entraîner un grand nombre de reconfigurations dans un environnement dynamique. Dans dhtFlex [18], les reconfigurations sont périodiques, ce qui peut entraîner la perte de tous les duplicatas si la fréquence des défaillances est trop élevée par rapport à la durée entre deux reconfigurations.

### 4.1.6 Conclusion

Offrir des services hautement disponibles aux utilisateurs de la grille est un enjeu majeur. Pour passer à l'échelle, les systèmes de grille sont en général composés d'un grand nombre de services. Nous proposons, au travers de Semias, un cadre pour mettre en œuvre des services de grille hautement disponibles et auto-réparants.

Semias est fondé sur l'utilisation de techniques de duplication active au dessus d'un réseau pair à pair structuré. La duplication active assure la haute disponibilité des services. Le réseau pair à pair structuré offre une solution de routage passant à l'échelle et tolérante aux fautes. De plus, il permet de prendre en charge un grand nombre de services et d'équilibrer automatiquement la charge induite par les services dupliqués sur l'ensemble des nœuds de la grille. Il n'existe pas à notre connaissance de mise en œuvre de duplication active au-dessus d'un réseau logique structuré.

Mettre en place de la duplication active dans un réseau pair à pair structuré soulève le problème de la gestion des reconfigurations. Pour un bon fonctionnement de Semias, deux propriétés doivent être préservées : i) les duplicatas d'un service doivent se trouver sur les nœuds du réseau logique les plus proches de la clé associée à ce service ; ii) le degré de duplication d'un service doit être maintenu. Chaque ajout ou retrait de nœud dans le réseau logique peut alors entraîner une reconfiguration dans un ou plusieurs groupes de duplicatas. Or reconfigurer un groupe de duplicatas est une action coûteuse, car impliquant des transferts d'état de services. Pour être auto-réparant, Semias doit être capable de prendre par lui-même les décisions de reconfiguration. Nous visons des mécanismes d'auto-réparation permettant d'assurer les deux propriétés mentionnées tout en limitant le nombre de reconfigurations des groupes de duplicatas.

Pour atteindre cet objectif, nous fondons notre solution sur l'architecture proposée par Mena et al. pour les systèmes de communication de groupes, qui permet de découpler suspicion d'un nœud et reconfiguration d'un groupe en plaçant le module de composition des groupes au dessus du module en charge de la diffusion atomique.

L'analyse de l'état de l'art sur la haute disponibilité dans les systèmes de grille existants nous permet de penser que de nombreux systèmes de grille actuels peuvent profiter de Semias.

## 4.2 Semias, un cadre pour la mise en œuvre de services hautement disponibles et auto-réparants dans la grille

Dans un premier paragraphe, nous décrivons les principes de fonctionnement de Semias en mettant en évidence les choix de conception qui permettent à Semias d'être adapté à

des environnements d'exécution dynamiques. Puis nous détaillons les modules composant l'architecture de Semias. Enfin nous présentons les mécanismes de gestion des reconfigurations et proposons un ensemble de règles pour limiter le nombre de reconfigurations intervenant dans le système, c'est-à-dire le nombre de changements dans les groupes de duplicatas, tout en assurant la haute disponibilité des services dupliqués.

### 4.2.1 Principes de fonctionnement

Semias est composé de quatre modules comme le décrit la figure 4.8. Les deux premiers modules sont le réseau logique structuré et le système de communication de groupes. Le réseau logique structuré offre une solution de routage passant à l'échelle et tolérante aux fautes. De plus, il permet d'associer les entités du système à des identifiants logiques plutôt qu'à des adresses physiques. Le système de communication de groupes fournit les primitives de diffusion atomique à un groupe de processus et permet donc la duplication active des services.

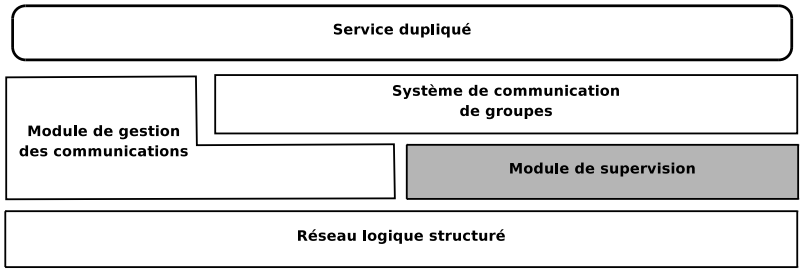



Fig. 4.8 – Les modules composant l'architecture de Semias

Le module de supervision est le module clé pour l'auto-réparation dans Semias. Il est chargé de regrouper les informations sur l'état des nœuds du système et sur l'état des groupes de duplicatas pour prendre les meilleures décisions de reconfiguration de groupes possibles. Son objectif est de limiter le nombre de reconfigurations tout en assurant que les services dupliqués continuent à être disponibles. Pour cela, il applique un ensemble de règles d'auto-réparation que nous avons définies pour décider de la nécessité d'une reconfiguration.

La détection des défaillances est un point crucial dans Semias. Les algorithmes de maintenance du réseau logique structuré sont utilisés comme détecteur de défaillances. Le réseau logique structuré génère des événements pour toutes les modifications, *i.e.* défaillances mais aussi arrivées de nouveaux nœuds, intervenant dans le voisinage d'un nœud. Ces événements sont ensuite exploités par les autres modules et notamment le module de supervision.

Enfin le module de gestion des communications a la charge des communications entre les clients et les services dupliqués. Il permet de rendre la duplication des services transparente pour les clients.

### 4.2.2 Description de l'architecture

Nous présentons maintenant en détail les quatre modules composant Semias. Une représentation détaillée de l'architecture est fournie par la figure 4.9.

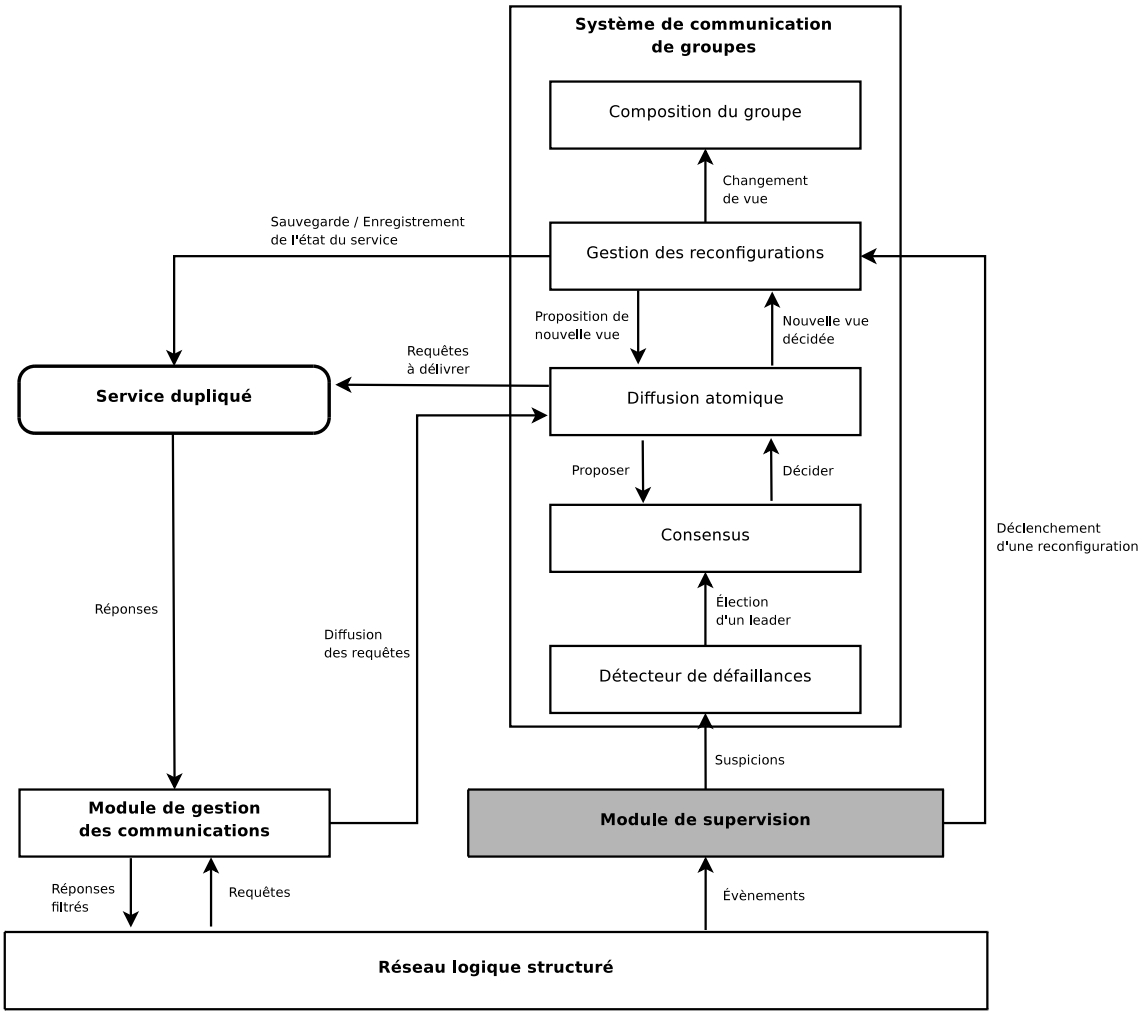



FIG. 4.9 – Architecture de Semias

#### 4.2.2.1 Le réseau logique structuré

Le réseau logique structuré employé dans Semias est fondé sur Pastry [163]. L'algorithme de maintenance des tables de routage employé est celui de Bamboo [157]. Dans Pastry, chaque nœud tient à jour une liste de ses $2 * L$ voisins, $L$ voisins sur la gauche et $L$ voisins sur la droite. Dans l'algorithme de maintenance de Bamboo, les voisins s'échangent périodiquement leur liste de voisins pour se tenir au courant des changements dans le voisinage et ainsi partager une vue cohérente. De plus, les voisins s'envoient régulièrement des messages de test pour détecter les défaillances. De plus, tout message envoyé à un voisin sert aussi de message de test. Nous définissons cinq événements associés aux changements pouvant intervenir dans le voisinage d'un nœud :

**Adhésion :** Pour intégrer le réseau pair à pair, un nœud associé à une clé $A$ doit contacter un nœud du réseau et router un message vers le nœud $B$ actuellement dans le réseau ayant la clé la plus proche de $A$. Le nœud $B$ ajoute $A$ dans sa liste de voisins et envoie cette liste à $A$. $A$ fait alors partie du réseau structuré. Au moment où $B$ ajoute $A$ dans sa liste de voisins, il génère un événement de type *adhésion(A)*.

**Arrivée :** Une fois $A$ intégré au réseau, ses voisins prennent connaissance de ce nouveau nœud grâce à l'échange périodique des listes de voisins. Lorsqu'un nœud ajoute $A$ à sa liste de voisins, il génère un événement de type *arrivée(A)*.

**Suspicion :** Si un nœud $D$ envoie un message au nœud A et n'obtient pas d'acquittement avant un temps défini comme *DélaiSuspicion*, le nœud $A$ est suspecté par $D$ d'être défaillant et un événement *suspicion(A)* est généré.

**Fin de suspicion :** Si le nœud $D$ parvient à communiquer à nouveau avec le nœud $A$ qui était précédemment suspecté, il génère un événement *finsuspicion(A)*.

**Défaillance :** Si après un délai *DélaiDéfaillance*, le nœud $D$ n'a toujours pas réussi à communiquer avec le nœud $A$, il enlève $A$ de sa liste de voisins et génère un événement *défaillance(A)*.

Tous ces événements sont traités par le module de supervision des groupes pour obtenir une vue de l'état du système. Les événements de suspicion sont utilisés pour fournir le détecteur de défaillance nécessaire à l'algorithme de consensus utilisé pour mettre en œuvre la diffusion atomique. Les événements de défaillances sont utilisés pour décider des reconfigurations des groupes de duplicatas. Comme nous allons le voir dans le paragraphe 4.2.2.3, *DélaiSuspicion* peut être beaucoup plus court que *DélaiDéfaillance*.

#### 4.2.2.2 Le module de gestion des communications

Le module de gestion des communications assure les communications entre les clients et les services dupliqués, c'est-à-dire qu'il gère les communications de *1 vers n* entités et de *n vers 1*.

Pour envoyer une requête à un service, un client se sert de la clé de ce service. Le module de gestion des communications du nœud hébergeant le client route le message en utilisant le réseau pair à pair. Le message est reçu par le nœud ayant la clé la plus proche de la clé du service. Ce nœud héberge normalement un duplicata de ce service. Le module de gestion des communications de ce nœud est alors chargé d'appeler la primitive de diffusion atomique pour diffuser la requête à l'ensemble des membres du groupe de duplicatas concerné.

Comme chaque modification dans le réseau pair à pair n'implique pas de reconfiguration immédiate des groupes de duplicatas, il est possible que le nœud recevant le message n'héberge pas encore de duplicata pour ce service. Comme nous l'expliquons en détail

dans le paragraphe 4.2.3, ce nœud doit alors être un nœud *retransmetteur* pour ce groupe de duplicatas. Le module de gestion des communications du nœud *retransmetteur* est chargé de transmettre la requête vers un nœud hébergeant actuellement un duplicata pour ce service.

Les services étant dupliqués activement, tous les duplicatas d'un service traitent les requêtes et envoient leur réponse au client. Comme nous supposons qu'il n'y a pas de faute byzantine dans notre système, le client peut prendre en compte la première réponse qui lui parvient. Le module de gestion des communications du nœud hébergeant un client est chargé de transmettre au client la première réponse à sa requête arrivant et d'éliminer les réponses suivantes.

#### 4.2.2.3 Le système de communication de groupes

Le système de communication de groupes met en œuvre la duplication active des services. Il est fondé sur l'architecture proposée par Mena et al. [128] et se divise en cinq modules. Un module de composition de groupe fournit une vue de la composition actuelle de groupe. Un module de diffusion atomique met en œuvre l'algorithme de diffusion atomique entre les membres du groupe. Cet algorithme utilise un module de consensus pour décider l'ordre des messages diffusés dans le groupe. Un détecteur de défaillances est nécessaire pour assurer la vivacité de l'algorithme de consensus. Enfin un module de gestion des reconfigurations est chargé de calculer la nouvelle vue du groupe lorsqu'une reconfiguration est déclenchée. Sur un nœud, il existe une instance du système de communication de groupe par duplicata hébergé par ce nœud. Nous détaillons dans la suite de ce paragraphe ces quatre modules.

**Diffusion atomique** Le module de diffusion atomique met en œuvre l'algorithme décrit par Schiper [167] pour la diffusion atomique de messages dans un groupe dynamique. C'est un algorithme fondé sur l'utilisation d'un algorithme de consensus impliquant tous les membres du groupe. Les messages sont ordonnés en utilisant une suite d'instances de l'algorithme de consensus exécutées de manière séquentielle. Chaque instance de consensus permet de décider un ensemble de messages à délivrer. Les messages qui sont ordonnés peuvent être de deux types, applicatifs ou relatifs à la composition du groupe. Dans un ensemble de messages décidé par consensus, sont d'abord délivrés les messages applicatifs concernant la vue courante du groupe, puis les messages relatifs à la composition du groupe qui entraînent un changement de vue.

Comme nous l'avons vu dans le paragraphe 4.1.5.2, un algorithme de diffusion atomique fondé sur un algorithme de consensus requiert un détecteur de défaillances de type $\diamond S$ [41], contrairement aux algorithmes fondés sur un séquenceur fixe ou tournant qui requièrent un détecteur de défaillance parfait [57]. Ainsi avec ce type d'algorithme, suspicion d'un nœud et éviction du nœud du groupe peuvent être découplées.

Cette propriété est une propriété clé pour Semias car, pour limiter le nombre de reconfigurations, il faut un algorithme de diffusion atomique capable de continuer à fonctionner même quand certains membres du groupe sont défaillants.

Dans Semias, *DélaiSuspicion* est beaucoup plus petit que *DélaiDéfaillance*. Les suspicions sont utilisées pour le détecteur de défaillance non fiable fourni à l'algorithme de consensus pour assurer une bonne réactivité de l'algorithme de consensus aux variations de performances des nœuds impliqués. Les événements de défaillances sont eux utilisés par le module de supervision pour décider des reconfigurations des groupes de duplicatas.

Illustrons ce fonctionnement par un exemple en prenant un *DélaiSuspicion* fixé à une

seconde et un *DélaiDéfaillance* fixé à une minute. Si un membre du groupe est retardé de quelques secondes, il va être suspecté et ne sera donc plus pris en compte par l'algorithme de consensus. Le retard de ce processus n'affecte donc pas les performances du consensus. Par contre, comme le processus n'est retardé que de quelques secondes, il ne sera pas considéré comme défaillant et ne sera donc pas retiré du groupe, évitant ainsi une reconfiguration inutile.

**Consensus** Pour choisir un algorithme de consensus, nous avons pris en compte les performances de différents algorithmes de consensus existants dans un environnement dynamique où peuvent intervenir plusieurs fautes et où certains processus peuvent être suspectés à tort. Dans ce contexte, les évaluations menées par Urban et al. [191, 192] sur trois des algorithmes de consensus les plus connus, *i.e.* Paxos [23], Chandra-Toueg [41], et Mostefaoui-Raynal [132], montrent que Paxos est le plus performant. C'est pourquoi nous avons opté pour cet algorithme. Paxos permet de décider une valeur quand au plus $n$ processus sont défaillants sur un ensemble de $2n + 1$ processus.

**Détecteur de défaillances** Le détecteur de défaillance requis par Paxos est $\Omega$ [40], détecteur de défaillances équivalent à $\diamond S$. $\Omega$ permet d'élire un *leader* : tous les processus non fautifs élisent ultimement le même processus non fautif comme *leader*.

Nous mettons en œuvre $\Omega$ en utilisant les événements *suspicion* et *finsuspicion* générés par les mécanismes de détection de défaillances du réseau pair-à-pair. Le leader dans un groupe de duplicatas, est le duplicata situé sur le nœud non suspect le plus proche de la clé du groupe.

**Module de gestion des reconfigurations** Le module de gestion des reconfigurations est chargé de décider de la composition de la nouvelle vue quand une reconfiguration est déclenchée. Une reconfiguration peut être déclenchée de deux manières. Le module de gestion des reconfigurations inspecte l'état de la configuration courante périodiquement à partir des informations fournies par le module de supervision pour voir si le nombre de duplicatas non fautif du groupe et leur position nécessitent une reconfiguration. Cependant si la fréquence des défaillances dans le système est élevée, on peut imaginer la perte de tous les duplicatas d'un groupe entre deux reconfigurations périodiques. C'est pourquoi le module de supervision déclenche une reconfiguration quand il estime que la configuration actuelle du groupe risque de compromettre la disponibilité du service dupliqué. Nous revenons en détail sur le problème des reconfigurations dans le paragraphe 4.2.3.

Quand une nouvelle configuration est nécessaire, le module de gestion des reconfigurations sélectionne les nœuds devant héberger un duplicata dans la nouvelle vue, à partir des informations fournies par le module de supervision. Il propose alors cette nouvelle configuration en utilisant la diffusion atomique. Les modules de gestion des reconfigurations de plusieurs duplicatas d'un groupe peuvent proposer une configuration pour la nouvelle vue. L'utilisation de la diffusion atomique assure que toutes ces propositions sont totalement ordonnées. Les vues sont identifiées de manière unique par un indice. Pour une vue donnée, seule la première proposition de configuration est prise en compte, assurant qu'une seule configuration peut être décidée pour une vue.

Quand une nouvelle vue a été décidée, le module de gestion des reconfigurations est chargé d'installer cette nouvelle vue, c'est-à-dire créer les nouveaux duplicatas et détruire les duplicatas qui n'appartiennent pas à la nouvelle vue.

Utiliser la diffusion atomique pour proposer une nouvelle vue assure que tous membres de la vue courante délivrent le même ensemble de messages avant de changer de vue. Seul

les processus appartenant à la vue courante peuvent participer à une instance de l'algorithme de consensus. Ainsi est assurée la sûreté de Paxos en dépit des reconfigurations.

**Module de composition du groupe** Le module de composition du groupe est un composant passif. Il fournit la composition du groupe pour la vue courante aux autres modules du système de communication de groupes. Il est mis à jour par le module de gestion des reconfigurations.

#### 4.2.2.4 Le module de supervision

Le module de supervision traite les événements générés par le réseau pair-à-pair. Chaque nœud du système a un module de supervision. Il tient à jour l'état, c'est-à-dire suspecté ou non, de chaque voisin du nœud et un journal contenant les dernières défaillances détectées dans le voisinage du nœud. Toutes ces informations peuvent être exploitées par le système de communication de groupe.

À chaque événement de type *arrivée* ou *défaillance* reçu, le module de supervision est chargé de vérifier si cet événement peut compromettre le bon fonctionnement des groupes dont un duplicata est hébergé sur ce nœud. Pour cela, il vérifie un ensemble de conditions que nous décrivons dans le paragraphe 4.2.3. Si tel est le cas, il notifie le module de gestion des reconfigurations du groupe concerné pour déclencher la reconfiguration de ce groupe.

### 4.2.3 Gestion des reconfigurations

Un enjeu majeur dans la mise en œuvre de stratégies de duplication dans un environnement dynamique tel qu'une grille de calcul est la gestion des reconfigurations. De nombreuses reconfigurations entraînent de nombreux et coûteux transferts d'état. Semias doit donc être capable de prendre des décisions de reconfigurations appropriées pour en limiter le nombre sans compromettre la disponibilité des services dupliqués. Dans ce paragraphe nous présentons notre solution pour atteindre cet objectif.

Nous commençons par introduire quelques termes qui sont utilisés par la suite. Puis nous décrivons le principe des nœuds retransmetteurs, indispensables pour éviter les reconfigurations à chaque arrivée d'un nouveau nœud dans la grille. Après avoir décrit les conditions exactes de sélection des nœuds devant héberger un duplicata d'un service, nous présentons un ensemble de règles que nous avons défini pour permettre à Semias de retarder la reconfiguration des groupes de duplicatas sans compromettre la disponibilité des services dupliqués. Enfin nous décrivons le processus de reconfiguration d'un groupe de duplicatas.

**Définition 4.4 (Nœud retransmetteur)** *Un nœud retransmetteur est un nœud susceptible de recevoir des messages destinés à un service dupliqué mais qui n'héberge pas de duplicata de ce service. Ce nœud transmet alors ces messages vers un nœud hébergeant un duplicata du service.*

#### 4.2.3.1 Définitions des termes utilisés

Nous considérons la *vue courante* d'un groupe de duplicatas. Lors d'une reconfiguration, nous passons de la *vue courante* à la *nouvelle vue* du groupe. Un *ancien duplicata* est un duplicata faisant partie de la *vue courante*. Un *nouveau duplicata* est un duplicata ne fait pas partie de la *vue courante* mais qui fait partie de la *nouvelle vue*. Les nœuds retransmetteurs qui ne sont pas inclus dans la nouvelle vue sont divisés en *anciens nœuds retransmetteurs*, qui ne sont plus des nœuds retransmetteurs après le changement de vue,

et en *nœuds retransmetteurs persistants*, qui restent des nœuds retransmetteurs après la reconfiguration.

#### 4.2.3.2 Les nœuds retransmetteurs

Ne pas reconfigurer les groupes de duplicatas à chaque arrivée d'un nouveau nœud dans le réseau pair-à-pair implique l'existence de nœuds retransmetteurs. En effet un nœud avec un identifiant plus proche de la clé d'un groupe de duplicatas que les nœuds hébergeant actuellement les duplicatas du groupe peut rejoindre le réseau pair-à-pair. Dans ce cas, les messages destinés à ce groupe de duplicatas vont être routés vers ce nœud dans le réseau logique. En attendant d'être intégré au groupe de duplicatas, ce nœud doit transmettre ces messages à un des duplicatas actuels du groupe : c'est un nœud retransmetteur.

Précisément, tout nœud joignant le réseau logique avec un identifiant plus proche de la clé d'un groupe de duplicatas qu'un des nœuds hébergeant actuellement un des duplicatas de ce groupe devient un nœud retransmetteur pour ce groupe. Ceci est illustré par la figure 4.10. Quand le nœud 34 rejoint le réseau logique, il doit devenir un nœud retransmetteur pour le groupe associé à la clé 44. Même si dans la configuration actuelle, il n'est pas supposé recevoir de messages pour le groupe 44 puisque le nœud 48 est plus proche de la clé que lui, il deviendra le nœud le plus proche si le nœud 48 subit une défaillance.

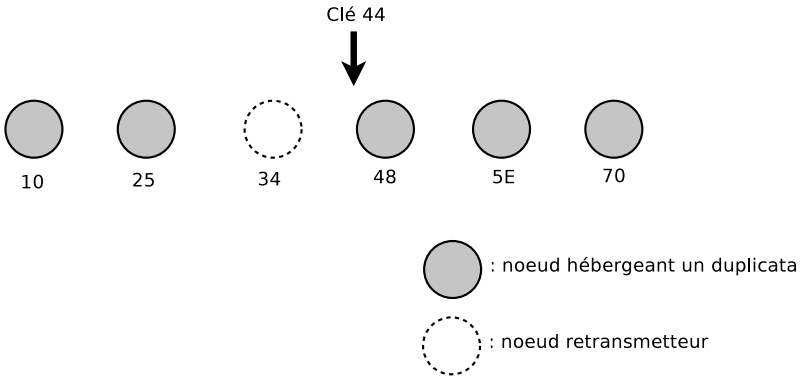



FIG. 4.10 – 5 duplicatas et 1 nœud retransmetteur

Un nœud retransmetteur doit connaître la composition actuelle des groupes pour lesquels il doit retransmettre les messages. Pour cela, quand un nouveau nœud $A$ veut intégrer le réseau logique, l'événement *adhésion(A)* généré par le nœud $B$ est intercepté par le module de supervision de $B$. Il contrôle alors si le nouveau nœud vérifie la condition d'un nœud retransmetteur pour un des groupes hébergé sur le nœud $B$ ou pour un des groupes pour lequel le nœud $B$ est lui-même un nœud retransmetteur. Si tel est le cas, est envoyé au nœud $A$ pour son initialisation, en plus de sa liste des voisins, la liste des membres non fautifs des groupes pour lesquels il devient un nœud retransmetteur. Quand un nœud retransmetteur reçoit un message pour un groupe pour lequel il n'héberge pas de duplicata, le module de gestion des communications du nœud se charge de transmettre le message à un des membres actuels du groupe.

#### 4.2.3.3 Position des duplicatas

Jusqu'à maintenant, nous avons considéré que les nœuds sélectionnés pour héberger les $n$ duplicatas d'un service étaient les $n$ nœuds ayant les identifiants les plus proches de la

clé du service. Cependant, pour assurer qu'un nœud intégrant le réseau, et devant devenir un nœud retransmetteur pour un groupe, contacte bien un nœud ayant connaissance de ce groupe, nous devons modifier cette règle comme le montre la figure 4.11.

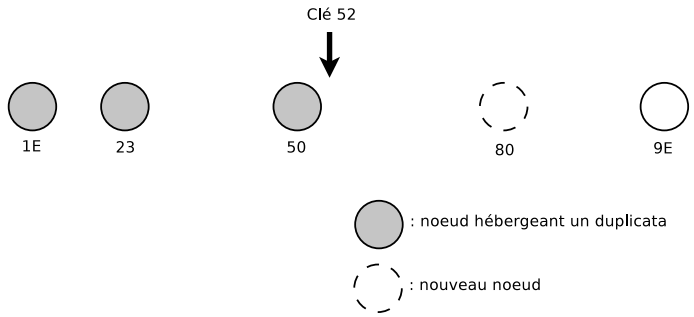



FIG. 4.11 – Exemple de mauvais positionnement des duplicatas

Sur cette figure, le degré de duplication est de trois. Les duplicatas du groupe 52 sont placés sur les trois nœuds les plus proches de la clé du groupe. Si un nœud rejoint le réseau logique avec l'identifiant 80, il doit être un nœud retransmetteur pour le groupe 52 car si le nœud 50 subit une défaillance, il devient le nœud le plus proche de la clé. Cepedant, pour se connecter au réseau, le nœud 80 contacte le nœud ayant l'identifiant le plus proche du sien, c'est-à-dire le nœud $9E$. Or le nœud $9E$ ne fait alors pas partie du groupe 52 et ne peut donc pas faire de 80 un nœud retransmetteur.

Pour éviter ce problème, la règle que nous fixons pour choisir les duplicatas d'un groupe est la suivante : la liste des $n$ nœuds hébergeant les duplicatas d'un groupe comprend le nœud le plus proche de chaque côté de la clé du groupe et les $n-2$ autres nœuds les plus proches de la clé du groupe.

#### 4.2.3.4 Règles de validité

Le module de gestion des reconfigurations examine périodiquement la configuration du groupe pour vérifier si des nœuds défaillants sont à enlever du groupe, ou si des nouveaux nœuds sont à ajouter au groupe, d'après la règle que nous avons établi précédemment sur le positionnement des duplicatas. Si un duplicata est défaillant, il doit être remplacé par un nouveau pour maintenir le degré de duplication du service. Si un nœud retransmetteur existe il doit être intégré au groupe pour que les requêtes envoyées au service dupliqué atteignent plus rapidement un membre du groupe. La fréquence de ces reconfigurations peut être faible car leur rôle est principalement d'optimiser le fonctionnement du groupe.

Cependant, si le nombre d'arrivées ou de défaillances de nœuds est important, la configuration d'un groupe pourrait devenir invalide entre deux vérifications périodiques. Par exemple, on peut imaginer la défaillance de tous les duplicatas d'un service entre deux vérifications périodiques. C'est pourquoi le module de supervision d'un nœud est chargé d'assurer la disponibilité de chaque groupe présent sur ce nœud, en vérifiant que la configuration actuelle des groupes respecte un ensemble de règles d'auto-réparation. Ces vérifications sont effectuées à chaque événement *arrivée* ou *défaillance* généré par le réseau logique. Nous présentons maintenant les trois règles qu'un groupe doit respecter et que le module de supervision est chargé de contrôler.

**Règle 4.1** *Il doit toujours y avoir une majorité de duplicatas non défaillants.*

L'algorithme de consensus a besoin d'une majorité de processus non défaillants pour pouvoir terminer. Quand le degré de duplication d'un groupe est de $2n + 1$, une reconfiguration doit donc avoir lieu si $n$ duplicatas sont détectés comme défaillants, pour intégrer de nouveaux duplicatas aux groupes.

**Règle 4.2** *Il doit toujours y avoir un duplicata de chaque côté de la clé du groupe.*

Cette règle est nécessaire aux mécanismes des nœuds retransmetteurs comme nous l'avons décrit dans le paragraphe 4.2.3.3. Dès que cette règle n'est plus vérifiée, une reconfiguration doit avoir lieu pour replacer les duplicatas dans une configuration valide.

**Règle 4.3** *Un nœud hébergeant un duplicata d'un groupe doit avoir dans sa liste de voisins l'ensemble des autres nœuds hébergeant des duplicatas pour ce groupe.*

Cette troisième règle est indispensable pour que le module de supervision soit capable d'assurer les deux premières. En effet, pour pouvoir superviser un groupe, le module de supervision doit recevoir tous les événements concernant les membres de ce groupe. Or un nœud du réseau logique ne génère des événements que pour les nœuds faisant partie de sa liste de voisins. L'arrivée de plusieurs nouveaux nœuds pourraient impliquer que deux nœuds hébergeant des duplicatas d'un même service, ne soient plus voisins comme l'illustre la figure 4.12. Dans cet exemple, supposons que la taille de la liste des voisins d'un nœud est de 16, 8 à gauche et 8 à droite. Dans ce scénario, l'arrivée de nouveaux nœud a pour conséquence de placer le nœud 10 hors de la liste des voisins de 70. Le module de supervision du nœud 70 ne sera donc pas averti si le nœud 10 subit une défaillance. Une reconfiguration doit donc avoir lieu avant que deux membres d'un groupe soit trop éloignés pour être voisins dans le réseau logique.

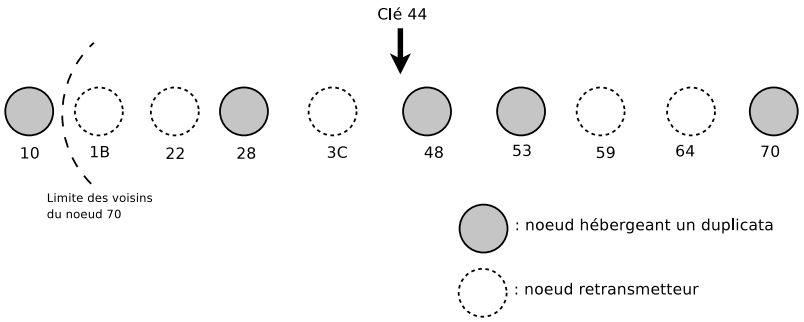



FIG. 4.12 – Scénario où la condition 4.3 n'est pas valide

L'hypothèse que nous faisons est que le temps nécessaire à la reconfiguration d'un groupe est suffisamment court pour que le risque qu'une nouvelle arrivée ou défaillance de nœud affecte le groupe avant que la reconfiguration n'ait eu lieu et rende le groupe invalide, est très faible. Dans le cas d'une grille très dynamique où cette hypothèse ne serait pas valide, il faudrait durcir les règles de validité pour que les reconfigurations aient lieu plus tôt. Nous présentons une évaluation de ce temps nécessaire à la reconfiguration d'un groupe dans le paragraphe 4.3.

#### 4.2.3.5 Procédure de reconfiguration d'un groupe de duplicatas

Dans ce paragraphe, nous décrivons la procédure de reconfiguration d'un groupe de duplicatas, depuis le moment où la reconfiguration est déclenchée jusqu'au moment où

tous les processus de la nouvelle vue sont prêts à délivrer des messages. La procédure de reconfiguration de Semias permet de changer tous les membres d'un groupe entre deux vues. De plus, elle garantit qu'aucun message applicatif n'est perdu lors d'un changement de vue.

La procédure de reconfiguration se divise en trois phases : (i) la nouvelle vue est proposée ; (ii) la nouvelle vue est installée ; (iii) les nouveaux duplicatas sont initialisés. Les communications entre les anciens et les nouveaux duplicatas durant cette procédure sont résumées par la figure 4.13. Sur cet exemple, la vue courante est composée de *Ancien duplicata 1*, *Ancien duplicata 2* et *Ancien duplicata 3*. Le nouvelle vue proposée par *Ancien duplicata 1* comprend *Ancien duplicata 2*, *Ancien duplicata 3* et *Nouveau duplicata*. De plus, dans cet exemple de changement de vue, nous avons deux nœuds retransmetteurs, un ancien et un persistant. Sur cette figure, certains messages n'ont pas été représentés en entier dans un souci de lisibilité.

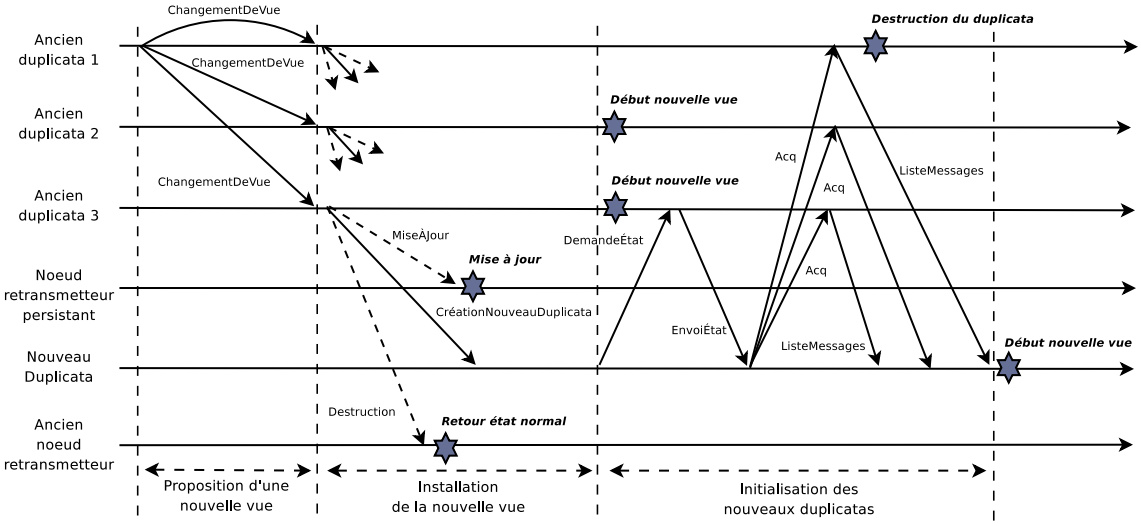



Fig. 4.13 – Exemple de changement de vue

**Proposition d'une nouvelle vue** La décision de changer de vue pour un groupe peut être prise par le module de supervision d'un des nœuds hébergeant un duplicata du groupe quand un événement met en péril la validité du groupe, où directement par le module de gestion des reconfigurations d'un des duplicatas quand il remarque lors de sa vérification périodique que certains duplicatas de la vue courante sont défaillants ou que leur position dans le réseau logique n'est plus celle désirée.

Quand la décision de reconfigurer est prise, le module de gestion des reconfigurations est chargé de calculer la nouvelle vue à partir des informations que lui fournit le module de supervision sur les défaillances et les arrivées dans le voisinage du nœud. Il diffuse ensuite sa proposition de nouvelle vue (message *ChangementDeVue*) en utilisant la diffusion atomique. Si plusieurs duplicatas proposent une nouvelle vue en même temps, seule la première proposition délivrée est prise en compte.

**Installation de la nouvelle vue** Le module de gestion des reconfigurations de tous les anciens duplicatas délivre le message *ChangementDeVue*. À cet instant débute l'installation de la nouvelle vue. Durant cette phase, aucun message applicatif n'est traité. Si

de nouveaux messages sont reçus, ils sont stockés temporairement par les duplicatas. Les anciens duplicatas commencent par sauvegarder l'état du service dupliqué pour pouvoir le fournir aux nouveaux duplicatas pour leur initialisation. Puis le module de gestion des reconfigurations met à jour l'état du module de composition du groupe avec la nouvelle vue.

En comparant la composition de la nouvelle vue à l'ancienne et à la liste actuelle des voisins du nœud, le module de gestion des reconfigurations détermine quels nœuds sont des anciens nœuds retransmetteurs et quels sont les nœuds retransmetteurs persistants. L'existence de nœuds retransmetteurs persistant entre deux vues, s'explique par le fait que le nœud qui a proposé la nouvelle vue n'avait pas connaissance de ces nœuds retransmetteurs au moment où il l'a fait et qu'ils n'ont donc pas été pris en compte dans le calcul de la vue. La raison peut être que ces nœuds ont rejoint le réseau logique juste avant la proposition de nouvelle vue voire même entre la proposition de nouvelle vue et son installation. Cependant s'ils sont toujours des nœuds retransmetteurs dans la nouvelle vue, cela signifie qu'ils sont toujours plus proches de la clé du groupe que certains membres de cette vue. Ils seront donc inclus dans le groupe lors de la prochaine reconfiguration.

Le module de gestion des reconfigurations de chaque ancien duplicata envoie un message de *Destruction* aux anciens nœuds retransmetteurs du groupe, et un message de *MiseÀJour* contenant la composition de la nouvelle vue aux nœuds retransmetteurs persistants pour qu'ils transmettent maintenant les messages aux membres de la nouvelle vue.

La phase d'installation se termine par l'envoi par chaque ancien duplicata d'un message *CreationNouveauDuplicata* vers tous les nouveaux duplicatas pour demander la création d'un nouveau duplicata. Ce message contient la nouvelle et l'ancienne vue, ainsi que les informations nécessaires à l'initialisation du consensus et de l'algorithme de diffusion atomique, *i.e.* le numéro de l'instance de consensus et le nombre de messages déjà délivrés au démarrage de la nouvelle vue. La liste des membres de l'ancienne vue permet aux nouveaux duplicatas de savoir qui contacter pour obtenir l'état du service dupliqué.

Quand la phase d'installation de la nouvelle vue est terminée, les anciens duplicatas appartenant à la nouvelle vue sont prêts à délivrer des messages pour cette nouvelle vue. Durant cette phase tous les envois de messages pour la procédure de reconfiguration sont effectués par tous les anciens duplicatas pour assurer l'absence de blocage de la procédure même en cas de défaillance de l'un d'entre eux.

**Initialisation des nouveaux duplicatas** Quand un nœud reçoit un premier message *CreationNouveauDuplicata* pour un groupe, il initialise l'instance du système de communication de groupes pour le nouveau duplicata à créer.

Il doit ensuite récupérer l'état du service dupliqué au près d'un des anciens duplicatas du groupe. Pour équilibrer la charge lors du transfert des états, chaque nouveau duplicata contacte l'ancien duplicata non défaillant ayant l'identifiant le plus proche du sien. Pour cela, le nouveau duplicata envoie un message *DemandeÉtat*. L'ancien duplicata lui envoie cet état dans un message *EnvoiÉtat*. Si l'ancien duplicata ne répond pas avant un certains délai, il est considéré comme défaillant et le nouveau duplicata contacte un autre ancien duplicata.

Après avoir initialisé son état, le nouveau duplicata envoie un acquittement (*acq*) à tous les anciens duplicatas. Sur réception de cet acquittement, chaque ancien duplicata envoie au nouveau duplicata un message *ListeMessages* contenant la liste des messages qu'il a reçus et mais qu'il n'a pas délivrés pour le service. Ainsi aucun message n'est perdu lors d'un changement de vue. Quand un nouveau duplicata a reçu la liste des messages

non délivrés de l'ensemble des anciens duplicatas non défaillants, il est prêt à participer à la nouvelle vue.

Quand un ancien duplicata a reçu un acquittement de tous les nouveaux duplicatas non fautifs, il peut détruire l'état du service sauvegardé au début de l'installation de la nouvelle vue puisque plus aucun nouveau duplicata n'en a besoin. Si le duplicata ne fait pas partie de la nouvelle vue, il peut être complètement détruit.

Il est important de signaler que durant toute la procédure de reconfiguration, le module de supervision de tous les nœuds concernés continue de contrôler les événements de défaillances pour éviter le blocage de la procédure.

## 4.3 Évaluation

Nous avons mis en œuvre un prototype de Semias que nous avons utilisé pour rendre le service de gestion des applications de Vigne hautement disponible et auto-réparant. Dans ce paragraphe, nous commençons par présenter notre prototype et son utilisation dans Vigne. Nous présentons ensuite une évaluation des performances des gestionnaires d'application de Vigne dupliqués. Nous commençons par une évaluation dans un environnement statique, c'est-à-dire sans défaillance et sans arrivée de nouveaux nœuds, puis dans un environnement dynamique. Enfin nous présentons une expérience menée à large échelle sur la grille.

### 4.3.1 Duplication active des gestionnaires d'application de Vigne

Ce paragraphe présente les points importants sur le prototype de Semias et décrit comment dupliquer un service en utilisant le prototype. Nous détaillons ensuite l'utilisation de Semias dans Vigne.

#### 4.3.1.1 Description du prototype de Semias

Le prototype de Semias est codé en C. Il est fondé sur la mise en œuvre de Pastry par Louis Rilling pour Vigne [103]. Semias est lancé en tant que processus *daemon* sur chaque nœud de la grille. Ce processus *daemon* exécute chaque service dupliqué dans un *thread* séparé.

Les algorithmes de routage de Pastry sont légèrement modifiés pour prendre en compte les nœuds suspectés de défaillance. Chaque nœud tente de transmettre les messages vers le nœud non suspect le plus approprié. De même, pour améliorer les performances, une requête n'a pas toujours besoin d'atteindre le nœud ayant l'identifiant le plus proche de sa clé. Si la requête transite par un nœud hébergeant un duplicata du service concerné, le module de gestion des communications de ce nœud peut directement diffuser atomiquement la requête dans le groupe de duplicatas sans transmettre le message vers le nœud ayant l'identifiant le plus proche de la clé du message.

L'algorithme de Paxos mis en œuvre est l'algorithme original [110]. De plus, pour des raisons de simplicité, les instances de consensus sont exécutées séquentiellement. Pour améliorer les performances, plusieurs instances de consensus pourraient être exécutées en parallèle et des optimisations telles que *Fast Paxos* [111] utilisées.

#### 4.3.1.2 Duplication d'un service avec Semias

Pour dupliquer un service existant en utilisant Semias, le programmeur doit tout d'abord mettre en œuvre les fonctions permettant de sauvegarder et de charger l'état

du service. Il doit ensuite modifier le service pour qu'il utilise les primitives de communication fournies par le module de gestion des communications de Semias. Du côté du client, la seule modification à apporter est de lui permettre d'adresser un service en utilisant une clé, au lieu d'une adresse physique. Au lieu de communiquer directement avec le service, un processus client communique alors avec le module de communication du *daemon* Semias du nœud sur lequel il s'exécute.

#### 4.3.1.3 Utilisation de Semias dans Vigne

Comme dans Vigne un réseau logique structuré était déjà utilisé pour localiser les gestionnaires d'application dans la grille et leur fournir des primitives de routage tolérantes aux fautes, la seule modification que nous avons eu à apporter fut de définir les fonctions permettant de charger et de sauvegarder l'état d'un gestionnaire d'application.

Dans la version actuelle du prototype, le degré de duplication des services dans Semias est constant. Le degré de duplication des gestionnaires d'application doit donc être choisi au démarrage du premier *daemon* Semias sur un nœud de la grille.

### 4.3.2 Évaluation dans un environnement statique

Pour évaluer l'impact de la duplication active sur un service, nous avons créé un client envoyant des requêtes de test à un gestionnaire d'application de Vigne. Ces requêtes ne nécessitent pas de traitement particulier de la part du gestionnaire d'application si ce n'est l'envoi d'un acquittement.

Nous évaluons tout d'abord le temps de réponse et le débit[2] pour un gestionnaire d'application dans un environnement statique, c'est-à-dire sans défaillance et sans arrivée de nouveaux nœuds.

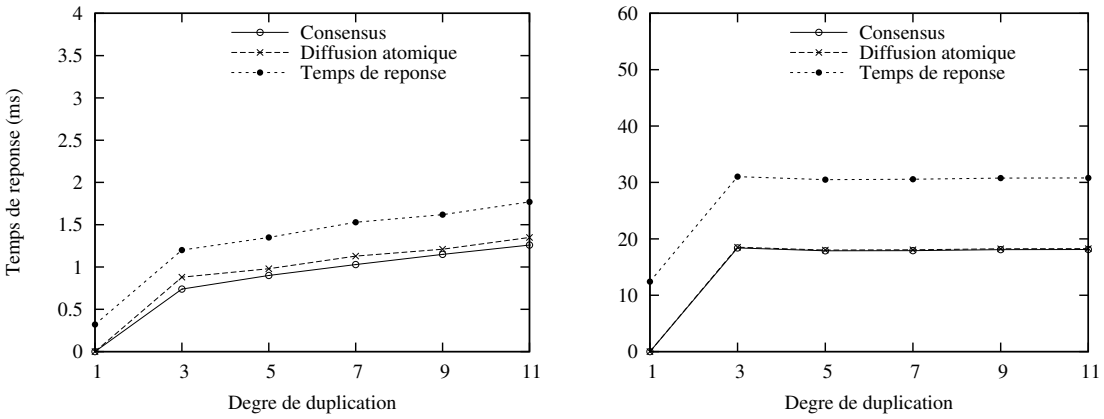
#### 4.3.2.1 Temps de réponse

Pour évaluer le temps de réponse, un client envoie une requête et attend la réponse avant d'envoyer la requête suivante. Les résultats présentés sont des moyennes sur 10000 requêtes. La figure 4.14 présente l'influence du degré de duplication sur le temps de réponse d'un gestionnaire d'application quand tous les duplicatas sont placés sur la même grappe de calcul (figure 4.14(a)), puis quand ils sont distribués sur la grille (figure 4.14(b)).

Pour l'expérience sur une grappe de calcul, nous choisissons des nœuds du site Grid'5000 de Rennes. Chaque nœud est équipé d'un processeur Intel Xeon à 2.33 GHz et de 4 GB de mémoire. Le client est exécuté sur un nœud n'hébergeant pas de duplicata.

La figure 4.14(a) montre l'évolution du temps de réponse du service en fonction du degré de duplication. Sur cette figure, nous montrons aussi le temps nécessaire à la diffusion atomique d'un message, et le temps nécessaire pour exécuter l'instance de consensus correspondante. Le temps nécessaire à la diffusion atomique est donc le temps nécessaire au consensus plus le temps nécessaire pour diffuser le message à l'ensemble des membres du groupe. Le temps de réponse est le temps nécessaire à la diffusion atomique plus le temps de transmission des messages entre le client et le service dupliqué. Cette figure montre que le temps de réponse est multiplié par trois entre un service non dupliqué et un service avec un degré de duplication de 3. Ce surcoût vient du temps nécessaire pour décider par consensus de l'ordre des messages. Le temps de réponse augmente ensuite linéairement avec le degré de duplication, car le *leader* du consensus doit toujours attendre une majorité de réponses avant de décider.

[2]Nombre de requêtes traitées par seconde




(a) Temps de reponse au sein d'une grappe

(b) Temps de reponse sur la grille

FIG. 4.14 – Temps de réponse d'un gestionnaire d'application avec différents degrés de duplication

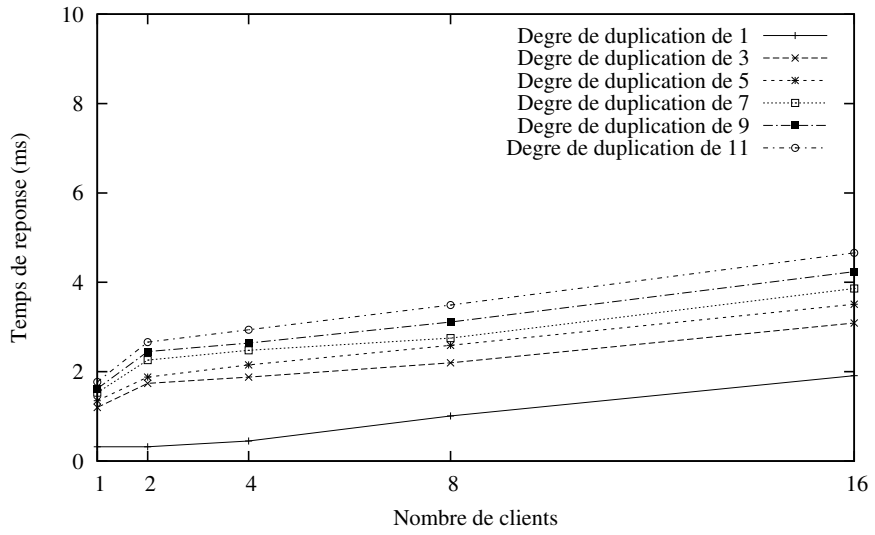



FIG. 4.15 – Temps de réponse d'un gestionnaire d'application avec plusieurs clients

La figure 4.15 présente l'évolution du temps de réponse d'un gestionnaire d'application pour différents degrés de duplication quand plusieurs clients effectuent des requêtes en même temps. L'augmentation du temps de réponse dû à la duplication s'explique par le fait que nous n'exécutons pas plusieurs instances du consensus en parallèle. Si une requête est reçue par le service et qu'une instance de consensus est déjà en cours d'exécution, il faut attendre la fin de cette instance avant de démarrer celle pour la nouvelle requête. Plus le nombre de clients est important, plus la probabilité que ce cas se produise est élevée.

Enfin la figure 4.14(b) montre l'évolution du temps de réponse en fonction du degré de duplication quand les duplicatas sont distribués sur la grille. Sur cette figure, les courbes représentant le temps nécessaire au consensus et le temps nécessaire à la diffusion atomique des messages sont confondus. Pour ces expériences, le *leader* du consensus est toujours situé à Grenoble et le client à Lille. Pour comprendre les résultats, le tableau 4.1 présente le positionnement des duplicatas pour chaque degré de duplication. Sur ce tableau est aussi présenté la latence entre les différents sites utilisés et Grenoble. Pour chaque degré de duplication, sont représentés en gras les duplicatas les plus proches du *leader* dont celui-ci doit attendre une réponse pour obtenir une majorité.

| **Site** | **Latence avec Grenoble** (ms) | **Degré 3** | **Degré 5** | **Degré 7** | **Degré 9** | **Degré 11** |
|---|---|---|---|---|---|---|
| Grenoble | 0.009 | **1** | **1** | **1** | **1** | **1** |
| Lyon | 2.4 | 0 | **1** | **1** | **2** | **3** |
| Sophia | 6.1 | **1** | 1 | 1 | 2 | 2 |
| Bordeaux | 6.1 | 1 | 1 | 1 | 1 | 2 |
| Orsay | 5.1 | 0 | **1** | **2** | **2** | **2** |
| Nancy | 7.0 | 0 | 0 | 1 | 1 | 1 |

TAB. 4.1 – Distribution des duplicatas sur 6 sites

La figure 4.14(b) montre que le temps de réponse n'augmente pas avec le degré de duplication. En effet, ici c'est la latence entre les duplicatas et le *leader* qui a le plus d'influence sur les performances et non pas le degré de duplication. Le temps de réponse de $30ms$ pour un service de grille nous semble tout à fait acceptable.

#### 4.3.2.2 Débit

Pour évaluer le débit d'un gestionnaire d'application dupliqué, c'est-à-dire le nombre de requêtes qu'il peut traiter par seconde, les duplicatas et le client sont positionnés de la même façon que pour l'évaluation du temps de réponse. Le client envoie maintenant 10000 requêtes de rang et attend ensuite les réponses.

La figure 4.16 montre le débit du service quand tous les duplicatas sont exécutés sur la même grappe de calcul puis quand les duplicatas sont distribués sur la grille. La baisse linéaire des performances sur une grappe de calcul s'explique par la majorité toujours plus grande que le *leader* doit attendre pour décider du consensus.

Sur la grille, le débit est divisé par 6 entre un gestionnaire d'application non dupliqué et un gestionnaire d'application dupliqué avec un degré de duplication de 3. Une nouvelle fois, cette baisse de performance est due au temps nécessaire pour le consensus. On observe à nouveau que les performances restent les mêmes quand on augmente le degré de duplication, ce qui s'explique par le positionnement des duplicatas dans la grille. Une nouvelle fois, nous pensons qu'un débit autour de 1000 requêtes par seconde est tout à fait acceptable pour un service de grille.

Les expériences en environnement statique montrent que le positionnement des duplicatas, *i.e.* la latence entre les duplicatas, a une influence plus importante sur les perfor-

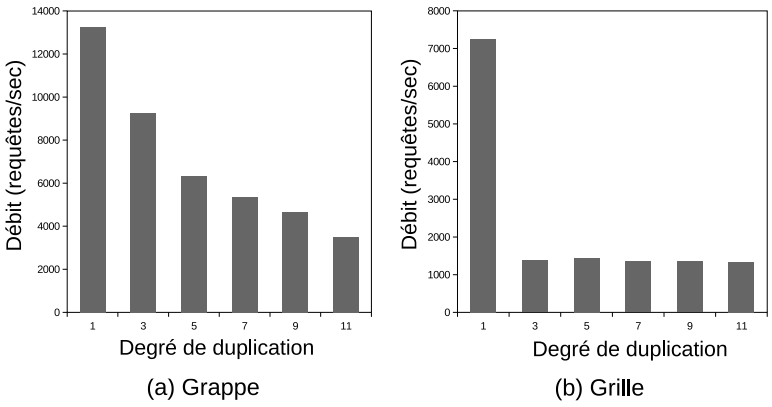



FIG. 4.16 – Débit d'un gestionnaire d'applications en fonction du degré de duplication

mances des services dupliqués que le degré de duplication. Il est donc possible d'utiliser un degré de duplication élevé pour assurer la disponibilité des services dans un contexte très dynamique. Cependant, avoir un grand degré de duplication implique aussi un coût important en terme de consommation de ressources puisque le nombre de services actif est plus important. Il est donc important de choisir un degré de duplication adapté au contexte d'exécution.

### 4.3.3 Évaluation dans un environnement dynamique

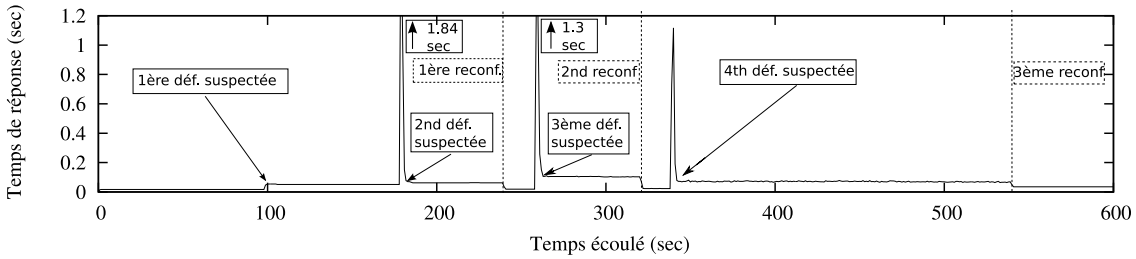
Pour évaluer les capacités d'auto-réparation de Semias, nous exécutons un gestionnaire d'application avec un degré de duplication de 5 dans la grille et évaluons son temps de réponse moyen. Les duplicatas sont distribués sur 4 sites de Grid'5000. Nous présentons tout d'abord un scénario d'exécution où certains duplicatas subissent des défaillances, puis un scénario où de nouveaux nœuds arrivent dans le réseau logique dans le voisinage des duplicatas. Pour évaluer la variation du temps de réponse, un client envoie une requête toutes les 100 ms. Les résultats présentés sont le temps de réponse moyen par seconde. La configuration de Semias est la suivante : la vérification périodique de la configuration du groupe par le module de gestion des reconfigurations est faite toutes les 300 secondes ; *DélaiSuspicion* est fixé à 3 secondes ; *DélaiDéfaillance* est fixé à 60 secondes.

#### 4.3.3.1 Scénario avec défaillances

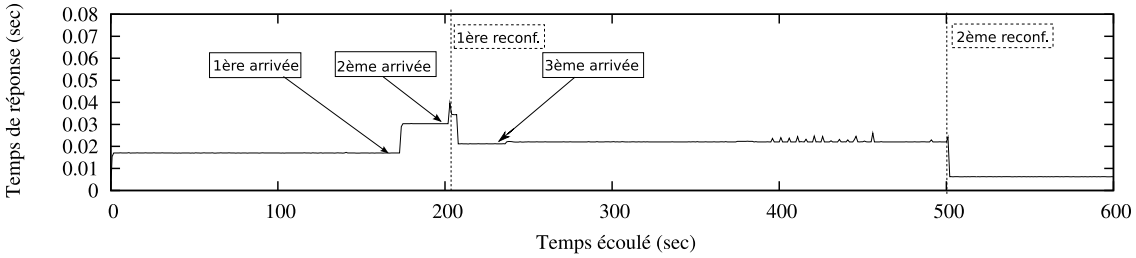
Pour mesurer l'impact des défaillances sur les performances d'un service dupliqué, nous *tuons* manuellement 4 nœuds hébergeant un duplicata à un intervalle de 80 secondes. La figure 4.17(a) montre l'évolution du temps de réponse au cours de ce scénario. Le temps écoulé est le temps depuis la première requête envoyée par le client.

3 reconfigurations ont lieu au cours de ce scénario. Les positions successives des duplicatas sont fournis par le tableau 4.2. Le premier duplicata subissant une défaillance est situé à Nancy. Les trois défaillances suivantes sont celles du duplicata le plus proche de la clé, c'est-à-dire successivement les deux duplicatas situés à Orsay puis celui situé à Bordeaux.

Le moment où une reconfiguration se produit est marqué par une ligne en pointillés sur la figure 4.17(a). La première et la troisième reconfiguration sont des reconfigurations périodiques. La première d'entre elles n'a pas lieu au bout de 300 secondes sur la figure




(a) Temps de réponse durant un scenario d'exécution avec défaillances




(b) Temps de réponse durant un scénario d'exécution avec arrivée de nouveaux noeuds

FIG. 4.17 – Variation du temps de réponse dans un environnement dynamique

| **Reconfiguration** | Duplicata 1 | Duplicata 2 | Duplicata 3 | Duplicata 4 | Duplicata 5 |
|---|---|---|---|---|---|
| 0 | Lyon | Orsay | **Orsay** | Nancy | Bordeaux |
| 1 | Lyon | **Orsay** | Orsay | Bordeaux | Lyon |
| 2 | Bordeaux | Lyon | **Bordeaux** | Lyon | Orsay |
| 3 | Nancy | Bordeaux | **Lyon** | Lyon | Orsay |

TAB. 4.2 – Position des duplicatas durant le scénario avec défaillances. Le duplicata en gras est le duplicata le plus proche de la clé du service pour chaque configuration.

simplement parce que le client n'a pas été démarré au même moment que le service dupliqué. Quand la première reconfiguration a lieu, la deuxième défaillance n'a pas encore été détectée. Le deuxième duplicata défaillant est donc inclus dans la nouvelle vue. C'est pourquoi lorsque la troisième défaillance est détectée, *i.e.* deux membres de la vue actuelle sont détectés comme défaillants, une reconfiguration est requise par la règle de validité 4.1.

Les défaillances peuvent conduire à une augmentation du temps de réponse pour trois raisons : (i) le temps nécessaire à l'élection d'un nouveau *leader* dans l'algorithme de consensus ; (ii) la gestion des liens de communication défaillants au niveau du réseau pair à pair ; (iii) la position des duplicatas non défaillants qui peuvent répondre au *leader* lors du consensus.

L'impact de la défaillance du *leader* de l'algorithme de consensus, *i.e.* le duplicata situé sur le nœud le plus proche de la clé du service, est visible lors des trois dernières défaillances. Chaque défaillance conduit alors à un pic du temps de réponse entre 1.11 and 1.84 secondes pour les messages envoyés au moment de la défaillance. Ce surcoût correspond au temps nécessaire pour suspecter le nœud défaillant. Tant que le nœud n'est pas suspecté, les messages destinés au service sont routé vers ce nœud dans le réseau logique. De plus, les autres duplicatas ne sont pas capables de traiter de messages puisqu'un nouveau *leader* n'a pas encore été élu pour le consensus. Quand le nœud est suspecté, un nouveau *leader* est élu et les messages sont routés vers le nœud non suspecté le plus proche de la clé.

Le coût de la gestion des liens de communication défaillants au niveau du réseau logique est visible après la première défaillance. Dans le réseau logique, un nœud qui doit envoyer un message à un nœud défaillant tente de recréer le lien de communication pour renvoyer ce message, jusqu'à ce que ce nœud sont considéré comme défaillant, *i.e.* après un délai *DélaiDéfaillance*. C'est la raison de ce surcoût.

L'impact de la position des duplicatas non défaillants est visible après la troisième défaillance. Ici le temps de réponse devient plus important car le nouveau *leader* est situé à Bordeaux et a donc une latence importante avec les autres duplicatas[3].

Cette expérience montre que même avec une fréquence de défaillance élevée, Semias assure la disponibilité des services. De plus, nous considérons que les surcoût observés sur le temps de réponse sont tout à fait acceptables. Le faible surcoût observé dans le cas où le *leader* subit une défaillance est lié au délai *DélaiSuspicion* que Semias permet de choisir petit.

#### 4.3.3.2 Scénario avec arrivée de nouveaux nœuds

Pour évaluer l'influence de l'arrivée de nouveaux dans le réseau logique dans le voisinage des duplicatas d'un service, nous exécutons un scénario où nous ajoutons successivement trois nœuds dont les identifiants sont choisis tels qu'ils deviennent des nœuds retransmetteurs pour le service : le nouveau nœud arrivant a toujours un identifiant plus proche de la clé que les nœuds déjà présents. L'intervalle entre deux arrivées est fixé à 30 secondes. La figure 4.17(b) montre la variation du temps de réponse du service au cours de ce scénario.

Deux reconfigurations ont lieu au cours de l'exécution. Les trois nouveaux nœuds sont respectivement positionnés à Bordeaux, Nancy et Orsay. Le tableau 4.3 résume les positions successives des duplicatas du service.

L'impact de l'arrivée d'un nouveau nœud n'est pas immédiat car il faut le temps qui celui-ci soit pris en compte au niveau des tables de routage dans le réseau logique. Jusqu'à

[3]Pour avoir le détail des latences entre les différents sites de Grid'5000, nous encourageons le lecteur à se référer au site *www.grid5000.fr*

| **Reconfiguration** | *Duplicata*1 | *Duplicata*2 | *Duplicata*3 | *Duplicata*4 | *Duplicata*5 |
|---|---|---|---|---|---|
| 0 | Nancy | Orsay | **Orsay** | Lyon | Lyon |
| 1 | Orsay | Bordeaux | **Nancy** | Orsay | Lyon |
| 2 | Orsay | Bordeaux | Nancy | **Orsay** | Orsay |

TAB. 4.3 – Position des duplicatas durant le scénario avec arrivée de nouveaux nœuds. Le duplicata en gras est le duplicata le plus proche de la clé du service pour chaque configuration.

ce moment, les messages sont toujours routés vers les anciens nœuds. C'est pourquoi on observe un délai entre l'arrivée du premier nœud et l'augmentation du temps de réponse. L'augmentation du temps de réponse s'explique par le fait que les messages arrivent d'abord sur le nouveau nœud qui doit les retransmettre vers un membre du groupe, et plus précisément vers le nœud du groupe ayant l'identifiant le plus proche de la clé. Les messages passent donc par un nœud intermédiaire supplémentaire avant d'atteindre un membre du groupe.

La seconde arrivée implique une reconfiguration imposée par la règle de validité 4.3. En effet, la taille de la liste des voisins d'un nœud dans le réseau logique est fixée à 16. La distance en nombre de nœuds entre deux duplicatas ne peut donc pas être supérieure 8. Cette reconfiguration a un impact sur le temps de réponse. En effet, les nouveaux nœuds étant plus proche de la clé du service que les nœuds hébergeant actuellement des duplicatas, l'un d'entre eux devient le *leader* du consensus après la reconfiguration. Plus exactement, c'est le dernier nœud arrivé qui devient le *leader*. Le pic correspond donc au temps nécessaire pour l'initialisation du nouveau *leader*. Comme la reconfiguration à lieu dès que l'arrivée de nœud, il n'est pas encore pris en compte au niveau des tables de routage lorsqu'il devient le *leader*. Durant cette période, les messages ne sont donc pas routés vers le *leader*, mais vers un des duplicatas qui appartenait déjà à la vue précédente. Ceci explique le palier observé juste après la reconfiguration : comme le leader ne reçoit pas directement les messages, il y a un délai supplémentaire avant qu'il ne démarre l'instance de consensus.

La deuxième reconfiguration est une reconfiguration périodique car l'arrivée du troisième nœud ne met en péril aucune règle de validité. Après cette reconfiguration, le temps de réponse est meilleur car maintenant une majorité des duplicatas est située sur le même site, Orsay.

Cette expérience montre que l'arrivée de nouveaux nœuds dans le voisinage des duplicatas d'un service, et que l'existence de nœuds retransmetteurs, induisent un surcoût très limité sur le temps de réponse d'un service dupliqué.

### 4.3.4 Passage à l'échelle

Pour évaluer les propriétés d'auto-réparation de Semias, nous effectuons une expérience à large échelle sur 100 nœuds de Grid'5000 répartis sur 7 sites. Nous démarrons 70 gestionnaires d'applications sur ces nœuds avec un degré de duplication fixé à 5. Les identifiants des nœuds et les clés des services sont choisis aléatoirement. La durée entre deux reconfigurations périodiques par Semias est fixée à 10 minutes. La fréquence moyenne d'arrivée et de défaillance de nœuds au cours de l'expérience est de 1 tous les 6 minutes. La durée totale de cette expérience est de 6 heures.

Le premier résultat est qu'à la fin de l'expérience tous les gestionnaires d'applications sont toujours disponibles. Au cours de l'expérience, 394 reconfigurations de groupes de duplicatas ont eu lieu. Durant l'expérience, nous avons aussi compté le nombre de recon-

figurations qui auraient eu lieu si les décisions de reconfigurations était prises à chaque arrivée ou défaillance de nœuds. Ce nombre est de 537. Les mécanismes d'auto-réparation de Semias, et en particulier l'utilisation de règles de validité, ont donc permis d'éviter 26% des reconfigurations tout en assurant la haute disponibilité des services.

## 4.4 Synthèse

Semias est une brique de base pour la conception de systèmes adaptés aux grilles de calcul de grande taille et dynamiques. Il permet, avec un minimum d'effort, de construire des services de grille hautement disponibles et auto-réparants.

Fondé sur la mise en œuvre de techniques de duplication active au-dessus d'un réseau logique structuré, il rend les défaillances et les reconfigurations totalement transparentes pour les utilisateurs du service. En effet, le réseau logique structuré offre des mécanismes de routage des messages tolérants aux fautes dans la grille, et permet de virtualiser la position des services qui sont adressables par des identifiants indépendant de leur position physique. La duplication active assure la disponibilité des services en dépit des défaillances.

Semias est un système auto-réparant. Les ajouts, défaillances et retraits de nœuds sont supervisés pour prendre les décisions de reconfiguration des groupes de duplicatas. Un module de supervision est chargé d'analyser la configuration du système et des groupes de duplicatas pour prendre les bonnes décisions de reconfiguration, permettant d'en limiter le nombre.

Semias est un système passant à l'échelle. Son architecture fondée sur l'utilisation d'un réseau logique permet de prendre en charge un grand nombre de services, et d'exploiter l'ensemble des nœuds d'une grille de calcul pour exécuter les services dupliqués.

Un prototype de Semias a été mis en œuvre en C. Ce prototype constitue à notre connaissance la première mise en œuvre de techniques de duplication active dans un réseau logique structuré.

Nous avons utilisé Semias pour faire du service de gestion d'applications de Vigne un service hautement disponible. Les expériences que nous avons menées sur la plate-forme d'expérimentation Grid'5000, nous ont permis de valider notre approche. Tout d'abord, nous avons montré que l'impact de Semias sur les performances des services dupliqués étaient acceptables. Nous avons ensuite montré que Semias parvenait à assurer la disponibilité des services dans un environnement très dynamique tout en conservant de bonnes performances. Enfin, nous avons montré que Semias, dans une expérience à large échelle, prenait des décisions d'auto-réparation permettant de limiter le nombre total de reconfigurations tout en assurant la disponibilité des services dupliqués

# Chapitre 5

# Tolérance aux fautes pour applications distribuées à échange de messages de grande taille

Dans ce chapitre, nous présentons O2P, un protocole de recouvrement arrière fondé sur l'enregistrement de messages optimiste *actif*. Il vise les applications distribuées de grande taille fondées sur le paradigme de communication par échange de messages.

L'enregistrement de message optimiste actif fait de O2P un protocole de recouvrement arrière passant à l'échelle. En enregistrant les informations de dépendance entre les processus de l'application au plus tôt sur support stable, O2P réduit les risques de création de processus orphelins. Surtout, il réduit la taille des informations de dépendances à attacher sur les messages de l'application en fonctionnement normal par rapport aux autres protocoles optimistes existants. Ainsi il limite le surcoût induit pas le protocole de recouvrement arrière sur les performances de l'application et assure un meilleur passage à l'échelle.

Pour améliorer le passage à l'échelle des protocoles à enregistrement de messages, nous proposons de plus dans ce chapitre, une solution pour la gestion distribuée de l'enregistrement des messages utilisant la mémoire vive des nœuds sur lesquels sont exécutés les processus de l'application.

O2P a été mis en œuvre dans la bibliothèque Open MPI et évalué expérimentalement sur Grid'5000. Les expériences mettent en évidence les propriétés de passage à l'échelle de O2P et de l'enregistrement distribué des informations de dépendance.

L'organisation de ce chapitre est la suivante. Dans le paragraphe 5.1, nous détaillons le modèle d'étude que nous considérons et mettons en évidence les limites des solutions de recouvrement arrière existantes en terme de passage à l'échelle. Dans le paragraphe 5.2, nous décrivons le protocole O2P et prouvons qu'il peut tolérer plusieurs fautes simultanées de processus. Le paragraphe 5.3 présente la mise en œuvre de O2P dans Open MPI et son évaluation sur Grid'5000. Il montre notamment les limites d'une solution centralisée pour l'enregistrement des messages et met en évidence les propriétés de passage à l'échelle de O2P associé à notre solution de sauvegarde distribuée des messages. Enfin nous dressons un bilan de ces travaux dans le paragraphe 5.4.

## 5.1 Problématique

Dans ce paragraphe, nous commençons par présenter le modèle d'application que nous considérons pour notre étude. Nous décrivons ensuite les enjeux liés à l'utilisation de

techniques de recouvrement arrière. Enfin, nous présentons un état de l'art détaillé des techniques de recouvrement arrière mettant en évidence les limites des solutions actuelles en terme de passage à l'échelle.

### 5.1.1 Modèle d'étude

Après avoir défini le modèle de système que nous considérons pour notre étude, nous définissons la relation d'ordre partiel entre les processus d'une application distribuée et précisons le modèle d'enregistrement de messages que nous utilisons pour cette étude.

#### 5.1.1.1 Système considéré

Nous considérons des applications distribuées composées d'un ensemble de processus communiquant en s'envoyant explicitement des messages. À chaque processus d'une application est associé un identifiant unique. Un canal de communication entre deux processus est FIFO[1], mais il n'y a pas d'ordre entre des messages envoyés sur des canaux de communication différents.

Le modèle de défaillance est celui que nous avons défini dans le paragraphe 1.2.2. Nous considérons des canaux de communication fiables et asynchrones mais il existe une borne maximale, potentiellement inconnue, sur le temps de propagation d'un message sur le réseau. Les processus subissent des défaillances par arrêt total. Une application peut subir des défaillances de plusieurs processus simultanément. Nous supposons l'existence d'un détecteur de défaillance dans le système. Enfin nous supposons qu'un service tel que XtreemGCP prend en charge le redémarrage des processus d'une application défaillante.

#### 5.1.1.2 Relation d'ordre partiel entre les processus d'une application distribuée

Les communications entre les processus d'une application distribuée créent des dépendances entre ces processus. L'émission et la réception de messages sont des événements. Dans un système asynchrone, il n'y a pas d'ordre de nature temporelle entre les événements survenant sur des processus différents. Cependant, Lamport [109] a défini une relation d'ordre partiel entre ces événements (*happened before relation*) notée $\prec$. $e \prec f$ signifie que l'événement $e$ précède causalement l'événement $f$, c'est-à-dire que $f$ dépend de $e$. Sur un même processus, $e \prec f$ si $e$ s'est produit avant $f$. Lors d'une communication, l'émission d'un message précède causalement sa réception.

Cette relation d'ordre partiel nous permet de redéfinir la notion de processus orphelin.

**Définition 5.1 (Processus orphelin)** *Un processus orphelin est un processus dont l'état courant dépend causalement d'un événement qui ne peut être rejoué après une défaillance.*

#### 5.1.1.3 Enregistrement de messages

Nous présentons en détails les principes des protocoles à enregistrement de messages dans le paragraphe 5.1.3.2. L'objectif des protocoles à enregistrement de messages est de sauvegarder les informations sur les événements non déterministes influençant l'exécution de l'application pour être capables de les rejouer après une défaillance.

Les travaux existants sur les protocoles à enregistrement de messages considèrent un modèle de communication générique où chaque réception de message est vue comme un événement non déterministe.

[1] « First In First Out »

Les travaux présentés dans ce chapitre sont mis en œuvre dans la bibliothèque Open MPI. Il a récemment été proposé un raffinement du modèle de communication dans le contexte des applications MPI permettant de mieux discriminer les événements non déterministes [27] associés aux réceptions de messages. En effet, l'analyse de la sémantique des primitives de communication définies par le standard MPI [71] montre que certaines d'entre elles conduisent en réalité à un comportement déterministe de l'application. Un protocole à enregistrement de messages n'a donc pas besoin de sauvegarder les événements associés à ces communications. Comme le nombre d'événements non déterministes à enregistrer est réduit, les performances des protocoles à enregistrement de messages sont meilleures. L'analyse des applications composant l'ensemble d'applications de tests du *NAS Parallel Benchmarks* [16], que nous utilisons pour nos évaluations, montre d'ailleurs que le nombre d'événements non déterministes associé à l'exécution de ces applications est très petit [27].

Cependant, dans ce chapitre nous utilisons le modèle générique car nous visons des solutions indépendantes du contexte MPI. Or le nouveau modèle ne s'applique qu'aux applications MPI. De plus, les applications dont nous disposons pour évaluer nos protocoles ne génèrent presque aucun événement non déterministe dans ce modèle. Or de nouvelles primitives MPI, telles que les primitives de communication collectives non bloquantes [89], ont été proposées pour améliorer les performances des applications MPI. Ces nouvelles primitives généreraient de nombreux événements non déterministes dans le nouveau modèle. C'est pourquoi nous estimons qu'évaluer nos solutions dans le nouveau modèle avec les applications dont nous disposons ne permettrait pas de tirer de conclusion sur l'utilité des solutions que nous proposons. Nous invitons le lecteur intéressé par une évaluation de nos solutions dans le nouveau modèle d'enregistrement de messages à lire l'article intitulé « *Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery* » [32].

### 5.1.2 Enjeux liés à l'utilisation de techniques de recouvrement arrière

Un protocole de recouvrement arrière doit répondre à plusieurs problèmes. La manière de répondre à ces problèmes permet de comparer les différentes solutions existantes :

**État global cohérent :** La première condition que doit remplir un protocole de recouvrement arrière est d'être capable après une défaillance de rétablir l'application dans un état global cohérent. Les techniques de recouvrement arrière diffèrent principalement dans les moyens choisis pour obtenir un état global cohérent après une défaillance. Selon la technique employée, l'état global cohérent obtenu peut correspondre à un état plus ou moins lointain dans la passé de l'exécution de l'application.

**Validation des sorties :** Lorsqu'une application communique avec le monde extérieur, il faut assurer qu'un message ne sera jamais invalidé par un retour arrière avant de l'envoyer. Le temps nécessaire à la validation d'un message à envoyer vers le monde extérieur est une caractéristique importante pour un protocole de recouvrement arrière.

**Performances :** Il est important de trouver un bon compromis entre performances en fonctionnement normal et performances au redémarrage. Les performances en fonctionnement normal dépendent du surcoût induit par le protocole de recouvrement arrière sur les performances de l'application lors d'une exécution sans défaillance. Les performances au redémarrage évaluent le temps nécessaire après une défaillance pour traiter les conséquences d'une défaillance. Ce temps se divise en deux phases :

(i) le temps nécessaire pour trouver l'état global cohérent et restaurer les processus dans cet état ; (ii) le temps nécessaire pour revenir dans un état équivalent à celui précédent la faute à partir de l'état global cohérent. Dans le cadre de notre étude, nous cherchons à obtenir un protocole avec le meilleur compromis pour des applications de grande taille.

### 5.1.3 État de l'art des techniques de recouvrement arrière

Nous avons énuméré dans le paragraphe 1.2.4.1 les principales familles de protocoles de sauvegarde de points de reprises existants. Nous revenons maintenant en détail sur chacune de ces familles, en en présentant les principes, les avantages et les inconvénients.

#### 5.1.3.1 Protocoles de sauvegarde de points de reprise

Ce paragraphe décrit les protocoles de recouvrement arrière fondés sur la sauvegarde de points de reprise.

**Protocoles de sauvegarde de points de reprise coordonnés** Les protocoles de sauvegarde de points de reprise coordonnés [42] coordonnent les processus de l'application avant la sauvegarde des points de reprise de processus pour assurer que l'état global sauvegardé soit cohérent. Nous avons présenté dans le paragraphe 3.2.4.1 un protocole coordonné qui vide les canaux de communications avant de sauvegarder les points de reprise pour assurer l'état global cohérent.

Le premier avantage des protocoles coordonnés est leur simplicité. De plus, l'assurance que le nouveau point de reprise coordonné constitue un état global cohérent permet de ne conserver que le dernier point de reprise de chaque processus. Pour valider un message à envoyer vers le monde extérieur, il suffit d'attendre le prochain point de reprise coordonné. Si la validation du message doit être faite rapidement, il faut alors forcer la sauvegarde d'un nouveau point de reprise coordonné.

La sauvegarde de points de reprise coordonnés a aussi de nombreux inconvénients. Tout d'abord, la défaillance d'un processus de l'application entraîne le retour arrière de tous les processus de l'application au dernier point de reprise. Ce comportement peut empêcher l'application de progresser quand la fréquence des défaillances est importante [117]. Plus le nombre de nœuds sur lesquels s'exécute une application est important, plus le risque de subir une défaillance est grand. Une grille de calcul offrant la possibilité d'exécuter des applications de très grande taille, cette limitation pose problème.

De plus, l'utilisation d'un protocole de sauvegarde de points de reprise nécessite une bonne gestion de l'écriture des points de reprise sur support stable. En effet, si tous les processus écrivent leurs points de reprise sur support stable au même moment, celui-ci peut devenir un point de contention surtout si il est mis en œuvre de manière centralisée [31]. Pour éviter ce problème, les points de reprises de chaque processus peuvent dans un premier temps être sauvegardés localement [117] puis être copiés de manière asynchrone sur support stable, les écritures pouvant alors être orchestrées de manière à limiter les accès concurrents au support de stockage stable. De plus, les points de reprise sauvegardés localement peuvent être utilisés pour améliorer les performances au redémarrage : les points de reprises étant disponibles localement pour les processus exécutés sur des nœuds non fautifs, seuls les processus fautifs ont besoin d'accéder au support de stockage stable au redémarrage.

**Protocoles de sauvegarde de points de reprise non coordonnés** Dans les protocoles de sauvegarde de points de reprise non coordonnés [20, 199], les points de reprise de processus sont sauvegardés de manière indépendante. L'état global cohérent n'est alors calculé qu'au moment du redémarrage après une défaillance.

Cette liberté peut permettre de décider localement du moment le plus opportun pour sauvegarder un point de reprise d'un processus, par exemple quand la taille de l'état du processus à sauvegarder est minimale. De plus, ce type de protocole permet de résoudre les problèmes d'accès concurrents au support stable lors de la sauvegarde des points de reprise.

Cependant, ils souffrent d'un problème important qui est l'effet domino, illustré par la figure 5.1 : il est possible qu'aucun état cohérent ne puisse être trouvé à partir de l'ensemble des points de reprise sauvegardés. Après la défaillance, le processus *p1* peut être redémarré à partir du point de reprise *pr1,1*. Les processus *p0* et *p1* doivent eux aussi effectuer un retour arrière car les messages *m5* et *m6* sont annulés par ce premier retour arrière. Cependant ni l'état formé par les points de reprise *pr0,0-pr1,1-pr2,1* ni celui formé par *pr0,0-pr1,0-pr2,0* ne sont cohérents. Dans le premier cas, le message *m4* serait orphelin. Dans le deuxième cas, c'est le message *m2* qui serait orphelin. Il faut donc redémarrer l'application depuis zéro. Le risque d'effet domino implique protocoles de points de reprise non coordonnés ne sont pas adaptés pour des applications ayant des interactions avec le monde extérieur.

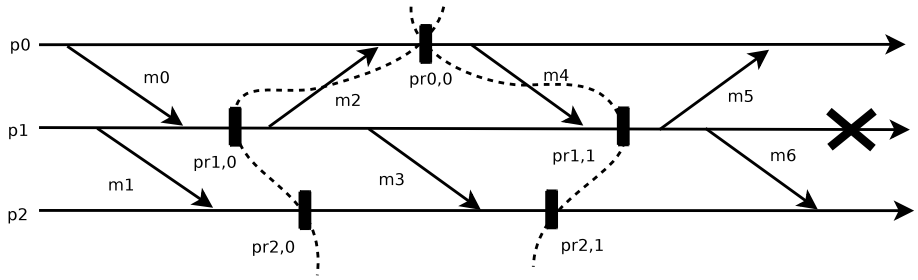



FIG. 5.1 – Sauvegarde de points de reprise non coordonnés

Comme l'état global cohérent n'est calculé qu'après une défaillance, il est nécessaire de conserver plusieurs points de reprise par processus. De nombreux points de reprise inutiles peuvent être sauvegardés, c'est-à-dire des points de reprise qui ne pourront jamais faire partie d'aucun état global cohérent, comme le point de reprise $pr0,0$ sur la figure 5.1.

**Protocoles de sauvegarde de points de reprise induits par les communications** Les protocoles de sauvegarde de points de reprise induits par les communications [87, 198] sont similaires aux protocoles non coordonnés dans la mesure où ils laissent la sauvegarde des points de reprise de processus se faire de manière indépendante. Cependant ils forcent des points de reprise additionnels pour éviter l'effet domino et assurer l'avancée de la ligne de recouvrement. Pour cela, les communications entre les processus sont supervisées pour détecter les motifs d'échange de messages qui peuvent conduire à la création de processus orphelins.

**Définition 5.2 (Ligne de recouvrement)** *La ligne de recouvrement est l'état cohérent dans lequel sera rétablie l'application si une défaillance intervient à un instant donné.*

Il a été montré que pour des applications de grande taille où les communications entre processus sont fréquentes, le nombre de points de reprise forcés devient très important, rendant ce type de protocole très coûteux [7].

#### 5.1.3.2 Protocoles à enregistrement de messages

**Principes** Les protocoles à enregistrement de messages supposent que l'exécution des processus d'une application distribuée est déterministe par morceaux [184], c'est-à-dire qu'elle est constituée d'une séquence d'intervalles d'exécution déterministes, simplement appelés intervalles d'exécution ($ie$), débutant chacun par un événement non déterministe, la réception d'un message. La figure 5.2 montre un processus dont l'intervalle d'exécution change à chaque réception de message.

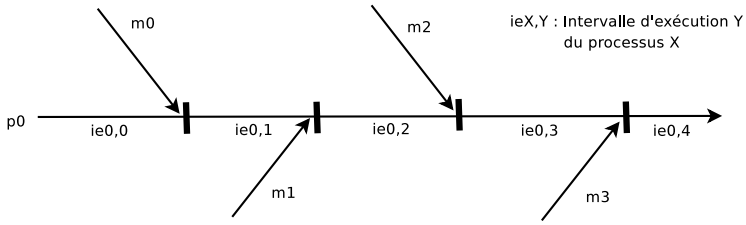



Fig. 5.2 – Intervalles d'exécution d'un processus

Deux processus déterministes par morceaux démarrant dans un état initial identique et délivrant la même séquence de messages, atteignent le même état. Pour restaurer un processus défaillant dans l'état précédent sa défaillance, il suffit donc de rejouer la même séquence de messages que celle reçu par le processus avant la défaillance. Pour cela, les protocoles à enregistrement de messages sauvegardent sur support stable, dans un journal, les déterminants associés aux messages délivrés par chaque processus. Un déterminant [6] décrit de manière unique un message. Il est composé des données contenues dans le message et d'un identifiant. Pour identifier les messages, chaque processus numérote les messages qu'il envoie avec un numéro d'émission ($Ne$) et les messages qu'il reçoit avec un numéro de réception ($Nr$). Ainsi l'identifiant d'un message est composé de l'identifiant de l'émetteur du message, de son numéro d'émission, de l'identifiant du récepteur du message et de son numéro de réception.

Après la défaillance d'un processus, il suffit donc de récupérer sur le support stable, la liste des déterminants des messages reçus par le processus avant la défaillance, pour pouvoir rejouer ces messages dans le même ordre et ainsi rétablir le processus dans l'état précédent la défaillance. Ainsi les protocoles à enregistrement de messages peuvent être combinés avec un protocole de sauvegarde de points de reprise non coordonnés sans risque d'effet domino. L'utilisation de points de reprise sert alors à réduire la taille des journaux et assure un redémarrage plus rapide de l'application après une défaillance.

Dans les protocoles à enregistrement de messages, les processus de l'application n'ont en général pas tous besoin d'effectuer un retour arrière après une défaillance. Les processus devant effectuer un retour arrière rejouent leur exécution à partir des déterminants sauvegardés sur support stable. Lors de ce rejeux, ils réémettent les messages qu'ils avaient déjà émis avant la défaillance. Les processus n'effectuant pas de retour arrière doivent alors être capables d'écarter ces messages pour ne pas délivrer plusieurs fois les mêmes messages.

Il existe trois familles de protocoles fondées sur l'enregistrement de messages : les protocoles optimistes, les protocoles pessimistes, et les protocoles causaux. Ils se différencient

par la manière de traiter la création de processus orphelins [6]. Avant de présenter ces trois familles de protocoles, nous présentons l'enregistrement de messages fondé sur l'émetteur qui est une optimisation pouvant être appliquée à tous les protocoles à enregistrement de messages.

**Enregistrement de messages fondé sur l'émetteur** L'enregistrement de messages fondé sur l'émetteur a été proposée par Johnson et Zwaenepoel [105]. Inclure le contenu des messages échangés par les processus de l'application dans les déterminants sauvegardés sur support stable peut être très coûteux. C'est pourquoi Johnson et Zwaenepoel proposent de sauvegarder le contenu des messages dans la mémoire volatile de l'émetteur. La taille des déterminants à sauvegarder sur support stable est alors considérablement réduite puisqu'ils ne sont plus composés que de l'identifiant des messages.

Le principe de l'enregistrement de messages fondé sur l'émetteur est simple. Si le destinataire d'un message subit une défaillance, le contenu de ce message est disponible dans la mémoire de l'émetteur pour le rejeux. Si l'émetteur subit lui aussi une défaillance, il doit rejouer son exécution et va alors regénérer les messages perdus.

Les évaluations menées par Elnozahy et Zwaenepoel [65] pour comparer l'enregistrement du contenu des messages sur support stable, aussi appelé enregistrement de messages fondé sur le récepteur, et l'enregistrement de messages fondé sur l'émetteur montrent clairement la supériorité de la seconde approche lors d'une exécution sans défaillance. L'évaluation de différents protocoles au redémarrage montrent que les performances des deux méthodes sont alors comparables [153]. Ces mêmes évaluations montrent d'ailleurs que les protocoles à enregistrement de messages ont des performances équivalentes quelque soit la famille à laquelle ils appartiennent lorsqu'ils sont combinés avec la technique d'enregistrement de messages fondé sur l'émetteur.

**Protocoles à enregistrement de messages pessimistes** Les protocoles à enregistrement de messages pessimistes [25, 105] assurent qu'aucune défaillance ne peut entraîner la création d'un processus orphelin. Pour cela les déterminants des messages sont sauvegardés de manière synchrone sur support stable : un processus ne peut envoyer un message que si les déterminants des messages qu'il a reçus précédemment ont été sauvegardés sur support stable. Comme le décrit la figure 5.3, certains envois de messages peuvent alors être retardés. La zone grisée sur la figure représente l'intervalle de temps durant lequel le processus $p1$ ne peut pas envoyer de message.

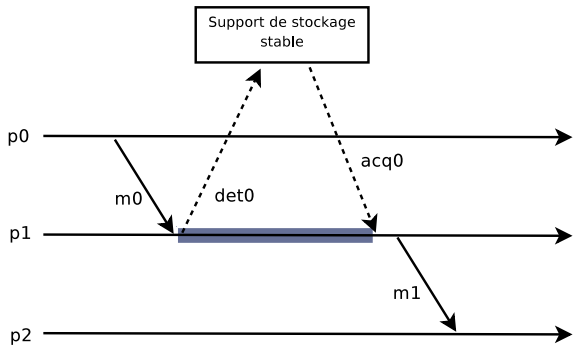



Fig. 5.3 – Protocole à enregistrement de messages pessimiste

La sauvegarde synchrone des déterminants assure qu'un message ne peut jamais de-

venir orphelin après la défaillance de son émetteur puisque ce dernier est alors capable de rejouer tous les messages qui l'avaient conduit dans l'intervalle d'exécution où le message a été émis. Ainsi lors d'une défaillance, seuls les processus fautifs doivent être redémarrés. Ces propriétés des protocoles pessimistes les rendent bien adaptés pour les applications communiquant avec le monde extérieur puisqu'un message envoyé au monde extérieur ne sera jamais invalidé.

Les évaluations menées dans le cadre de la librairie MPI tolérante aux fautes MPICH-V [30] sur des protocoles pessimistes [26, 33] montrent clairement que quand les communications sont fréquentes entre les processus de l'application, le surcoût engendré par l'enregistrement synchrone des déterminants sur support stable est très important.

**Protocoles à enregistrement de messages optimistes** Les protocoles à enregistrement de messages optimistes [53, 148, 173, 178, 184] sauvegardent les déterminants sur support stable de manière asynchrone pour obtenir de meilleures performances lors du fonctionnement normal de l'application. Comme le montre la figure 5.4, le processus *p1* peut envoyer le message *m1* sans attendre que le déterminant du message *m0* soit sauvegardé.

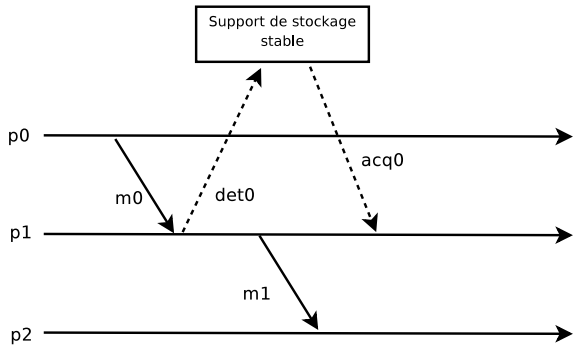



FIG. 5.4 – Protocole à enregistrement de messages optimiste

Cependant, sauvegarder les déterminants de manière asynchrone entraîne un risque de création de processus orphelins. En effet, si lors d'une défaillance, un déterminant est perdu, les processus dépendant causalement du message associé à ce déterminant deviennent orphelins. Il est alors nécessaire de mettre en œuvre un protocole de recouvrement arrière pour détecter les processus orphelins et rétablir l'application dans un état global cohérent. Les processus fautifs et les processus orphelins doivent effectuer un retour arrière après une défaillance. L'hypothèse optimiste associée aux protocoles optimiste décrite par la propriété 5.1, implique que le risque de création de processus orphelins est très faible. C'est pourquoi, dans la plupart des cas, seuls les processus fautifs ont un retour arrière à effectuer.

**Propriété 5.1 (Hypothèse optimiste des protocoles optimistes)**
*L'enregistrement d'un déterminant sur support stable est suffisamment rapide pour que le risque de subir une défaillance entre la réception d'un message et l'enregistrement du déterminant correspondant soit faible.*

Pour être capable de détecter les processus orphelins, les protocoles optimistes doivent tenir à jour les relations de dépendance entre les intervalles d'exécution des processus de l'application. Pour cela, des informations de dépendance doivent être attachées sur les

messages applicatifs. Les protocoles à enregistrement de messages optimistes existants se différencient principalement par la manière de tracer les dépendances entre les processus. Pour être capable de tolérer plusieurs défaillances, des vecteurs de dépendances [184, 173] ou des vecteurs d'horloge tolérants aux fautes sont utilisés [53, 177]. Dans tous les cas, la taille de ces vecteurs est proportionnelle à la taille de l'application.

Pour valider les messages à envoyer vers le monde extérieur, les processus de l'application doivent être tenus informés des intervalles d'exécutions qui deviennent stable sur les autres processus. La solution couramment utilisée est que chaque processus informe périodiquement les autres processus de son intervalle d'exécution stable maximal [53, 173].

**Définition 5.3 (Intervalle d'exécution stable)** *Un intervalle d'exécution d'un processus est stable lorsque les déterminants de tous les messages reçus par ce processus avant cet intervalle d'exécution ont été sauvegardés sur support stable.*

**Définition 5.4 (Intervalle d'exécution valide)** *Un intervalle d'exécution d'un processus est valide quand tous les intervalles d'exécution dont il dépend sont stables.*

Sur l'exemple de la figure 5.5, l'intervalle d'exécution *ie1,2* est stable. Cependant il n'est pas valide car il dépend de l'intervalle d'exécution *ie0,1* qui n'est pas stable. Un message ne peut être envoyé vers le monde extérieur que lorsqu'il est certain qu'il ne deviendra jamais orphelin, *i.e.* lorsque son intervalle d'exécution d'émission est valide.

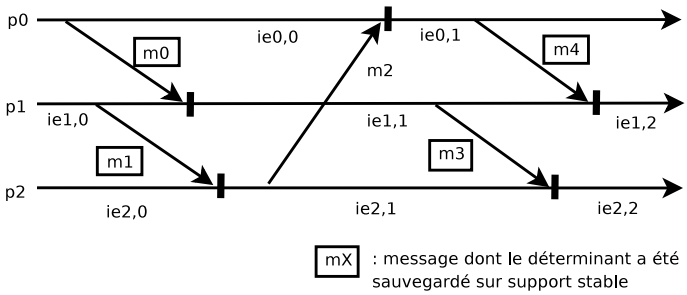



Fig. 5.5 – Exemple d'état stable et d'état valide

**Protocoles à enregistrement de messages causaux** Les protocoles à enregistrement de message causaux [6, 64, 115] empêchent la création de processus orphelins sans retarder l'émission des messages. Pour cela, ils attachent sur chaque message envoyé les déterminants desquels dépend ce message, comme l'illustre la figure 5.6. Ici le déterminant *det0* est attaché au message *m1* et sauvegardé en mémoire volatile par le processus *p2*. Ainsi quand un processus subit une défaillance, les informations nécessaires pour rejouer l'exécution de ce processus sont disponibles dans la mémoire des processus non fautifs dépendant de ce processus. Dans l'exemple de la figure 5.6, si le processus *p1* subit une défaillance, le déterminant du message *m0* est disponible dans la mémoire du processus *p2*.

Les protocoles causaux sont présentés comme optimaux d'un point de vue théorique [6] car ils assurent que seul les processus fautifs ont à redémarrer après une défaillance, sans retarder l'émission des messages lors du fonctionnement normal de l'application. Pour pouvoir envoyer un message vers le monde extérieur, un processus doit simplement sauvegarder les informations de dépendance de son état courant sur support stable avant d'envoyer son message.

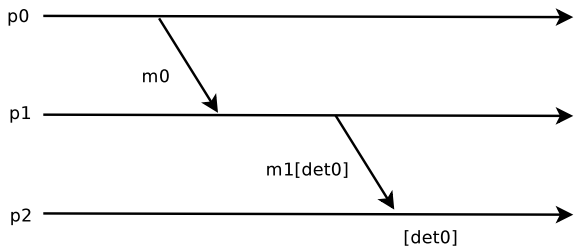



FIG. 5.6 – Protocole à enregistrement de messages causal

Le principal surcoût engendré par les protocoles causaux est lié à la gestion des informations de dépendance entre les processus de l'application [28]. Pour savoir quelles informations doivent être attachées sur le prochain message émis par un processus, ce processus doit tenir à jour une structure de données représentant ses relations de dépendance causale [64, 115]. Cette structure de donnée doit être parcourue au moment de l'émission d'un message pour obtenir la liste des déterminants à attacher sur le message et mise à jour lors de la réception d'un message avec les informations de dépendance attachées à ce message. Le surcoût ainsi engendré devient important quand la taille de l'application et la fréquence des messages augmente [28] car la quantité d'information attachée sur chaque message devient alors très grande.

Pour réduire la quantité d'information à attacher sur chaque message, il a été proposé de sauvegarder les déterminants aussi sur support stable [28] : quand un déterminant est sauvegardé sur support stable, il n'a plus besoin d'être attaché sur les messages émis par un processus. Cette optimisation permet de réduire l'impact des protocoles causaux sur les performances en fonctionnement normal des applications. Cependant quand la taille des applications augmente et que la fréquence des échanges de message est élevée, le surcoût engendré en fonctionnement normal reste élevé.

### 5.1.4 Conclusion

Notre objectif est de proposer une solution de recouvrement arrière pour des applications à échange de messages de grande taille. Les techniques fondées sur la sauvegarde de points de reprise coordonnés ne sont pas adaptées pour ce type d'application car la défaillance d'un processus a pour conséquence le retour arrière de l'ensemble des processus de l'application, ce qui peut empêcher l'application de progresser. L'utilisation de points de reprise non coordonnés combinée avec un protocole à enregistrement de messages semble être une bonne alternative.

La technique d'enregistrement de messages fondée sur l'émetteur permet de limiter le coût de l'enregistrement de messages. L'évaluation de performances au redémarrage des différentes familles de protocoles à enregistrement de messages dans ce cadre montre des résultats équivalents. C'est pourquoi nous nous concentrons sur les performances les protocoles à enregistrement de messages en fonctionnement normal.

Les protocoles à enregistrement de messages pessimistes sont fondés sur l'enregistrement synchrone des déterminants sur support stable, ce qui implique un surcoût important sur les performances en fonctionnement normal des applications.

Les protocoles à enregistrement de messages optimistes et causaux évitent cette synchronisation avec le support stable. Le principal surcoût en fonctionnement normal vient alors de la quantité de données à attacher sur chaque message de l'application.

La taille des informations à attacher sur les messages applicatifs pour un protocole optimiste est moins importante que pour un protocole causal. En effet, un protocole causal doit attacher toutes les informations nécessaires pour pouvoir rejouer un message alors qu'un protocole optimiste doit seulement attacher les informations nécessaires à la détection des processus orphelins. Ceci est illustré par le simple exemple de la figure 5.7. Dans cet exemple, un protocole causal doit attacher sur le message $m3$ les déterminants des messages $m1$ et $m2$ dont dépend $m3$. Dans la même situation, un protocole optimiste doit simplement attacher l'information précisant que le message $m3$ a été envoyé durant l'intervalle d'exécution 2 du processus $p0$ pour être capable de savoir si le processus $p2$ devient orphelin en cas de défaillance de $p0$.

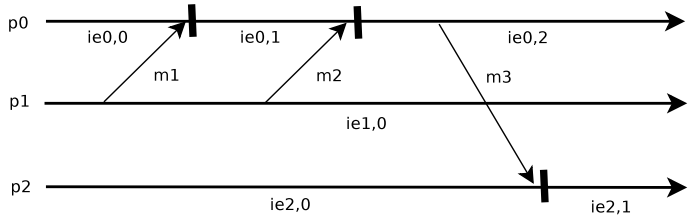



Fig. 5.7 – Simple scénario d'exécution

C'est pourquoi nous proposons l'utilisation d'un protocole à enregistrement de messages optimiste pour la tolérance aux fautes d'applications distribuées à échange de messages de grande taille. Les protocoles optimistes existants capables de tolérer plusieurs fautes [53, 173, 177, 184] attachent sur chaque message applicatif un vecteur de dépendances ou un vecteur d'horloge dont la taille est proportionnelle à la taille de l'application. Nous voulons proposer un protocole à enregistrement de messages optimiste capable de prendre en compte les informations déjà sauvegardées sur support stable pour diminuer la taille des informations à attacher sur les messages applicatifs et ainsi obtenir un protocole passant à l'échelle.

## 5.2 O2P, un protocole à enregistrement de messages optimiste actif

O2P est un protocole à enregistrement de messages optimiste. Ils se fondent sur l'enregistrement de message optimiste actif pour réduire la taille des informations à attacher les messages de l'application par rapport aux autres protocoles optimistes existants.

### 5.2.1 Influence de la quantité de donnée attachée sur les messages d'une application

Pour illustrer l'intérêt de l'enregistrement optimiste actif, nous présentons une expérience mesurant l'impact des données attachées sur les messages applicatifs sur les performances d'une application. Pour cela nous utilisons l'application NetPIPE [179] qui permet de mesurer la latence et la bande passante entre deux machines sur un test de type *Ping-Pong*. Les deux machines utilisées pour ce test sont équipées d'un processeur Intel Xeon 5148 LV, de 8 GB de mémoire et sont reliées par un lien Gigabit Ethernet sur un Switch Cisco 6509.

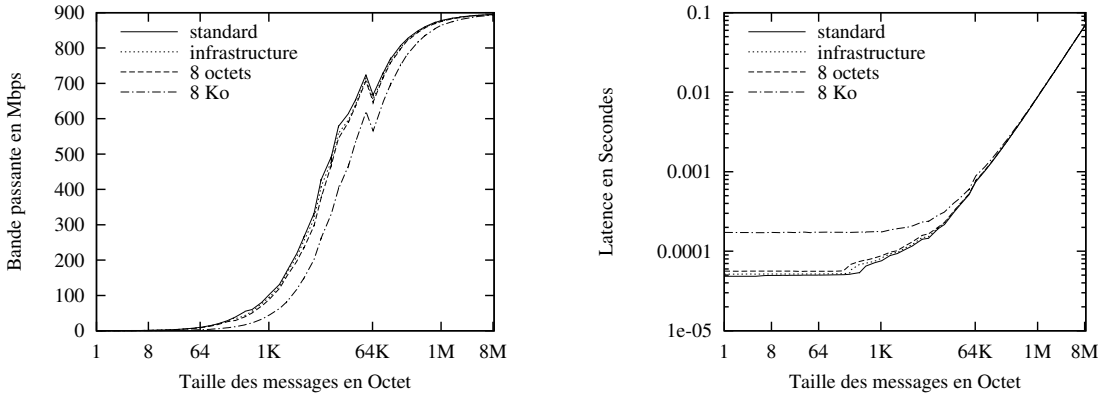



FIG. 5.8 – Surcoût engendré par l'ajout de données sur les messages applicatifs

Dans cette expérience, nous comparons les performances de Open MPI[2], représentées par la courbe nommée *standard*, avec les performances lorsque respectivement 8 octets et 8K octets de données sont attachés sur chaque message. Comme nous le décrivons dans le paragraphe 5.2.4.1, un intervalle d'exécution est représenté par un indice et un numéro d'incarnation. En supposant que ces deux valeurs sont stockées dans deux entiers et qu'un entier est codé sur 4 octets, 8 octets sont nécessaires pour identifier un intervalle d'exécution. Ainsi 8K octets représentent la taille d'un vecteur de dépendance pour une application composée de 1024 processus, qui est l'ordre de grandeur des applications que nous visons. La courbé nommé *infrastructure* représente le coût des mécanismes que nous utilisons pour attacher des données sur les messages applicatifs. Ces mécanismes sont décrits dans le paragraphe 5.3.2.2.

Quand 8K octets de données sont attachés sur chaque message, la figure 5.8 montre que l'impact sur la latence est important surtout pour des messages de taille comprise entre 1 octet et 64K octets : le surcoût est alors de l'ordre de 250%. L'impact sur la bande passante est surtout notable pour des messages de taille comprise entre 64 octets et 1M octet : il est alors de l'ordre de 15%. Il est intéressant de noter que pour des petits messages, avoir 8 octets de données supplémentaires attachées, implique déjà un surcoût sur la latence supérieur à 15%.

Le tableau 5.1 montre la répartition des tailles de messages pour les applications du *NAS Parallel Benchmark*, qui regroupe un ensemble d'applications représentatives des applications de calcul scientifique. La répartition des tailles de message observée montre que réduire la quantité de données attachées par un protocole optimiste sur les messages de l'application peut clairement contribué améliorer les performances du protocole en fonctionnement normal.

### 5.2.2 Principes de l'enregistrement de messages optimiste actif

Ce paragraphe décrit le principe de l'enregistrement de messages optimiste actif. Nous commençons par mettre en avant les similarités et les limites des protocoles à enregistrement de messages optimistes existants. Ils se concentrent tous sur l'optimisation des performances au redémarrage des applications après défaillance mais pour cela attachent sur chaque message des données dont la taille est proportionnelle à la taille de l'application. Nous présentons ensuite l'enregistrement de messages optimiste actif et montrons

[2]Révision 22041 de http ://svn.open-mpi.org/svn/ompi/trunk

| Application | Nombre de message par processus (Fréquence) | 0-64 | 64-1K | 1K-64K | 64K-1M |
|---|---|---|---|---|---|
| BT.B.64 | 9743 (309 msg/s) | 0.8 % | 0.1 % | 86.7 % | 12.4 % |
| CG.B.64 | 19995 (1360 msg/s) | 60.4 % | 0.0 % | 0.0 % | 39.6 % |
| FT.B.64 | 566 (109 msg/s) | 4.9 % | 0.3 % | 0.0 % | 94.8 % |
| LU.B.64 | 88822 (5569 msg/s) | 0.1 % | 98.9 % | 0.0 % | 1.0 % |
| MG.B.64 | 3335 (3705 msg/s) | 32.8 % | 32.4 % | 34.7 % | 0.0 % |
| SP.B.64 | 19342 (460 msg/s) | 0.4 % | 0.1 % | 87.0 % | 12.5 % |

TAB. 5.1 – Nombre et taille des messages en octets émis par les applications du NAS Parallel Benchmark (Classe B, 64 processus)

comment notre protocole O2P peut tirer avantage de ce mécanisme pour réduire la taille des informations à attacher sur les messages de l'application. Nous présentons aussi les autres avantages que peut avoir cette solution.

#### 5.2.2.1 Les protocoles optimistes existants

L'objectif principal poursuivi par les protocoles optimistes capables de tolérer plusieurs fautes qui ont été proposés dans la littérature, est d'optimiser les performances lors du redémarrage après défaillance. Il s'agit alors de limiter au maximum les retours arrière suite à une défaillance et d'assurer une gestion asynchrone de ces retours arrière. Pour cela, ces protocoles se sont concentrés sur la manière de représenter et de tracer les dépendances entre les intervalles d'exécution des processus de l'application pour détecter efficacement les processus orphelins. La technique la plus courante est alors d'utiliser des vecteurs d'horloge [126] auxquels sont ajoutées des informations sur les défaillances de processus par l'intermédiaire de numéros d'incarnation, le numéro d'incarnation d'un processus étant incrémenté à chaque fois que celui-ci redémarre. Ces vecteurs sont alors appelés vecteurs d'horloge tolérants aux fautes [53] ou plus simplement vecteurs de dépendances [54, 148, 173, 184]. Une autre solution consiste à distinguer le temps à différents niveaux [177, 178], *i.e.* le temps du point de vue applicatif et le temps du point de vue du protocole de tolérance aux fautes. Des vecteurs d'horloge différents sont alors utilisés pour représenter ces niveaux de temps.

Dans tous ces protocoles, la taille des vecteurs attachés sur les messages applicatifs est proportionnelle à la taille de l'application. Cependant les résultats présentés dans [153] ainsi que dans le paragraphe 5.2.1 nous laisse penser que le point le plus important pour obtenir un protocole à enregistrement de messages optimiste performant, n'est pas d'optimiser les performances au redémarrage mais d'optimiser les performances en fonctionnement normal en limitant la taille des informations à attacher sur les messages applicatifs.

Une solution a été proposée pour tolérer plusieurs fautes avec un protocole optimiste sans attacher d'information sur les messages applicatifs [195]. Cependant le nombre d'échange de messages nécessaires pour trouver un état global cohérent après une défaillance augmente très rapidement avec la taille de l'application. De plus, comme les

dépendances entre les processus ne sont pas tenues à jour durant l'exécution normale de l'application, il est impossible de savoir quand un message à envoyer vers le monde extérieur peut être validé.

L'objectif poursuivi par O2P, notre protocole fondé sur l'enregistrement de messages optimiste actif, est de limiter la taille des informations attachées sur les messages tout en conservant les bonnes propriétés des protocoles optimistes.

#### 5.2.2.2 Enregistrement actif des déterminants

Dans les protocoles à enregistrement de messages optimistes, les déterminants des messages délivrés par un processus sont temporairement stockés dans la mémoire du nœud sur lequel s'exécute le processus et sont de temps en temps[3] sauvegardés sur support stable.

Le principe de l'enregistrement actif des déterminants est d'enregistrer les déterminants sur support stable le plus tôt possible : dès qu'un message est délivré, le déterminant correspondant est envoyé vers le support stable. Les déterminants ne sont pas temporairement stockés dans la mémoire du nœud. Cette stratégie a plusieurs avantages :

**La quantité de donnée à attacher sur les messages de l'application est réduite.** Damani et al. [54] ont démontré que pour être capable de détecter les processus orphelins, il suffit de tracer les dépendances par rapport aux intervalles d'exécution non stables. En effet, après la défaillance d'un processus, seuls les intervalles d'exécution non stables de ce processus ne peuvent pas être restaurés. Les messages émis lors de ces intervalles d'exécution sont donc orphelins. Connaître les dépendances d'un processus par rapport aux intervalles d'exécution non stables est donc suffisant pour déterminer si un processus est orphelin. L'enregistrement de message optimiste actif assure que les intervalles d'exécution des processus deviennent stables au plus tôt. Cette propriété peut être exploitée pour limiter la taille des données à attacher sur les messages applicatifs par un protocole optimiste.

**Les risques de pertes de déterminants sont réduits.** En cas de défaillance d'un processus, les déterminants stockés en mémoire sont perdus. Plus les déterminants restent longtemps en mémoire, plus la probabilité de les perdre est grande. C'est pourquoi sauvegarder les déterminants au plus tôt limite les risques de pertes de déterminants et donc, limite les risques de création de processus orphelins.

**La validation des messages vers le monde extérieur est plus rapide.** Comme les déterminants sont sauvegardés plus rapidement, les intervalles d'exécution des processus deviennent stables plus rapidement, ce qui est important pour la validation des messages à envoyer vers le monde extérieur.

### 5.2.3 Description du protocole O2P en fonctionnement normal

Dans ce paragraphe, nous décrivons O2P, notre protocole de tolérance aux fautes fondé sur l'enregistrement de messages optimiste actif. Nous détaillons tout d'abord les hypothèses optimistes faites par ce protocole. Puis nous mettons en évidence que la gestion des sauvegardes sur support stable est un point critique dans le bon fonctionnement de O2P. Enfin, nous fournissons une description détaillée du protocole en fonctionnement normal.

[3] La sauvegarde des déterminants sur support stable peut être périodique ou déclenchée quand le nombre de déterminants dépasse une certaine limite.

#### 5.2.3.1 Un protocole optimiste

Le nom « *O2P* » signifie « *Optimist Optimist Protocol* ». En effet, O2P est fondé sur deux hypothèses optimistes. La première hypothèse est celle classiquement faite par les protocoles à enregistrement de messages optimistes et énoncée par la propriété 5.1. La deuxième hypothèse sur laquelle se fonde O2P est la suivante :

**Propriété 5.2 (Hypothèse optimiste du protocole O2P)** *L'enregistrement d'un déterminant sur support stable est suffisamment rapide pour que la probabilité d'avoir sauvegardé ce déterminant avant l'envoi du prochain message soit grande.*

Si cette deuxième hypothèse est toujours vérifiée, comme sur la figure 5.9, alors aucune donnée n'a besoin d'être attachée sur les messages de l'application par le protocole. En effet, dans cet exemple l'intervalle d'exécution d'émission du message *m1* ne dépend d'aucun déterminant non sauvegardé sur support stable au moment de l'envoi de *m1*, puisque le déterminant *det0* a pu être sauvegardé. Comme il n'est nécessaire de tracer les dépendances que par rapport aux intervalles d'exécution non stables, il n'y a ici aucune dépendance à tracer. Quand l'hypothèse optimiste du protocole O2P est valide, aucune donnée n'est attachée sur les messages applicatifs. Nous prouvons dans le paragraphe 5.2.4.5 que les informations attachées par O2P sur les messages applicatifs sont suffisantes pour trouver les processus orphelins après une défaillance en un temps limité.

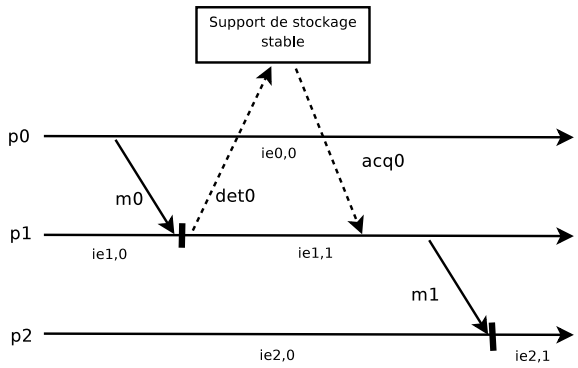



FIG. 5.9 – Hypothèse optimiste de O2P valide

Cependant, même l'enregistrement actif des déterminants ne peut assurer que la propriété 5.2 soit toujours vérifiée. Ceci est illustré par la figure 5.10. Ici le message *m1* est envoyé par le processus *p1* avant d'avoir été informé que le déterminant *det0* ait été sauvegardé. L'intervalle d'exécution *ie1,1* n'est donc pas stable au moment de l'envoi. Cette information doit être attachée sur le message *m1*. Quand le processus *p2* délivre ce message, il met à jour ses dépendances en ajoutant une dépendance par rapport à l'intervalle d'exécution *ie1,1*. Ce type de scénario risque de se produire quand les processus de l'application communiquent de manière fréquente.

Sur l'exemple de la figure 5.10, si le processus *p2* doit envoyer un message dans l'intervalle d'exécution *ie2,1*, il devra à nouveau y attacher la dépendance par rapport à l'intervalle d'exécution *ie1,1*. Cependant ici, même si *det0* a été sauvegardé sur support stable entre temps, le processus *p2* n'a pas connaissance de cette information. Ceci met en évidence la nécessité de mettre en place des mécanismes pour diffuser les informations sur les nouveaux intervalles d'exécution qui deviennent stable sur les processus de l'appli-

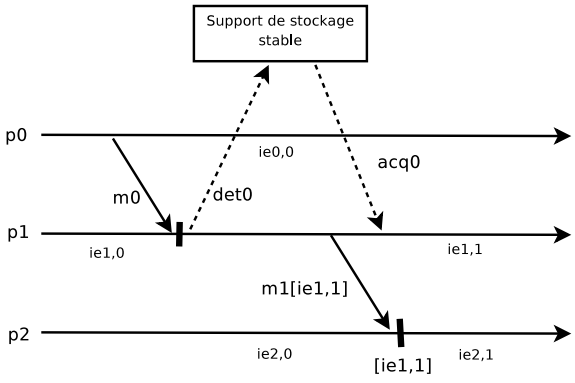



FIG. 5.10 – Hypothèse optimiste de O2P non valide

cation. Ces mécanismes pourront aussi être exploités pour valider les messages à envoyer vers le monde extérieur.

#### 5.2.3.2 Enregistrement des déterminants

Les figures 5.9 et 5.10 mettent en évidence que les performances de O2P sont liées aux performances de l'enregistrement des déterminants sur support stable. Si l'enregistrement des déterminants sur support stable n'est pas assez rapide, la quantité d'information à attacher sur les messages applicatifs ne peut être réduite.

Comme nous l'avons vu dans le paragraphe 5.1.3.2, l'enregistrement de message fondé sur l'émetteur est une technique bien connue pour réduire la quantité de données à sauvegarder sur support stable. Le contenu des messages peut être sauvegardé dans la mémoire de l'émetteur. Ainsi seul l'ordre de réception des messages doit être sauvegardé sur support stable.

Réduire la taille des déterminants à sauvegarder sur support stable permet de réduire le coût de cette sauvegarde. Ainsi dans notre protocole, les déterminants sauvegardés sur support stable sont simplement composés de l'identifiant des messages, c'est-à-dire l'identifiant de l'émetteur du message, le numéro d'émission, l'identifiant du récepteur du message et le numéro de réception. Nous y ajoutons le numéro de l'intervalle d'exécution d'émission du message pour les besoins de notre protocole. Les données composant un déterminant à sauvegarder sur support stable sont résumés par la figure 5.11.

1: **Définition de** $det_{msg}$**, déterminant du message** $msg$
2: $det_{msg}.source \leftarrow msg.source$, // *Identifiant du processus émetteur*
3: $det_{msg}.Ne \leftarrow msg.Ne$, // *Numéro d'émission du message*
4: $det_{msg}.iee \leftarrow msg.iee$, // *Numéro de l'intervalle d'exécution d'émission du message*
5: $det_{msg}.dest \leftarrow msg.dest$, // *Identifiant du processus destinataire*
6: $det_{msg}.Nr \leftarrow msg.Nr$ // *Numéro de réception du message*

FIG. 5.11 – Définition d'un déterminant à sauvegarder sur support stable

Dans les paragraphes suivants, nous décrivons en détail le fonctionnement de O2P. Dans cette description, nous présentons le support de stockage stable comme une *boite noire*. Tous les processus de l'application y ont accès et peuvent y sauvegarder des données. Un acquittement est utilisé pour confirmer l'enregistrement d'une donnée. Dans un premier temps, nous supposons que cet acquittement est aussi utilisé pour informer des nouveaux

intervalles d'exécution qui deviennent stable sur les différents processus de l'application. L'acquittement contient donc un vecteur, nommé *VecteurStable* dont l'entrée $i$ contient l'intervalle d'exécution stable maximum du processus $p_i$. Ce fonctionnement est résumé sur la figure 5.12.

```
1: Initialisation des variables
2:     VecteurStable ← [⊥, ..., ⊥]
3:
4: Sur réception d'un déterminant det du processus p_i
5:     Sauvegarder det
6:     VecteurStable[i] ← det.Nr
7:     Envoyer VecteurStable au processus p_i
```

FIG. 5.12 – Enregistrement de déterminants sur support stable

L'étude de la mise en œuvre d'un support de stockage stable dans le cadre de O2P est décrite dans le paragraphe 5.3.

#### 5.2.3.3 O2P en fonctionnement normal

Lors de l'exécution normale de l'application, O2P doit sauvegarder les déterminants des messages reçus par les processus de l'application pour, en cas de défaillance, pouvoir rejouer ces messages. Il doit, de plus, tenir à jour les dépendances entre les intervalles d'exécution des processus de l'application pour être capable de détecter les processus orphelins. La figure 5.13 décrit le comportement de O2P en fonctionnement normal. La version du protocole présentée ici est une version simplifiée qui ne prend pas en compte les problèmes relatifs à la gestion des défaillances. Une version complète du protocole est présentée dans le paragraphe 5.2.4.

**Tracer les dépendances entre les intervalles d'exécution** Chaque processus tient à jour deux vecteurs de taille $n$, *i.e.* le nombre de processus de l'application. Le vecteur nommé $VecteurDeDependances_i$ permet de sauvegarder les dépendances par rapport aux intervalles d'exécution non stables de l'intervalle d'exécution courant du processus $p_i$. Le vecteur nommé $VecteurStable_i$ sauvegarde l'état stable maximum de chaque processus de l'application, connu par le processus $p_i$.

Lorsqu'un processus envoie un message, il lui attache son vecteur de dépendances courant qui inclut toutes les informations sur les intervalles d'exécution non stables dont dépend le message envoyé (ligne 11).

Quand le processus reçoit ce message, il doit mettre à jour son propre vecteur de dépendances (ligne 24-29). Pour cela, il commence par enregistrer l'intervalle d'exécution créé par la réception du message dans son vecteur de dépendances (ligne 24). Puis il met à jour ce vecteur à partir du vecteur attaché sur le message. Pour cela, il prend en compte les intervalles d'exécution stables qu'il connaît déjà (ligne 26). Pour finir, il envoie le déterminant associé au message reçu au support stable pour qu'il soit sauvegardé (ligne 31).

Un processus met son vecteur de dépendances à jour lorsqu'il reçoit un nouveau $VecteurStable$ comme acquittement pour l'écriture d'un déterminant sur support stable (ligne 34-36). Si des intervalles d'exécution dont il dépend sont devenus stables, il peut les enlever de son vecteur de dépendances. Ici nous nous appuyons sur les canaux de communication FIFO et considérons donc que les déterminants sont sauvegardés dans l'ordre. Ainsi lorsque le $VecteurStable$ reçu indique que l'intervalle d'exécution $v$ du processus $p_i$

1: **Initialisation des variables du processus** $p_i$
2: $VecteurDeDependances_i \leftarrow [\perp, ..., \perp]$ // *Vecteur de dépendances*
3: $VecteurStable_i \leftarrow [\perp, ..., \perp]$ // *Vecteur des états stables maximums connus*
4: $MessageExterieur_i \leftarrow \perp$ // *Liste de messages à envoyer vers le monde extérieur*
5: $ie_i \leftarrow 0$ // *Intervalle d'exécution*
6: $Ne_i \leftarrow 0$ // *Numéro d'émission*
7:
8: **Envoi d'un message** $msg$ **au processus** $p_j$
9: $Ne_i \leftarrow Ne_i + 1$
10: Initialiser $msg$ // $msg.source \leftarrow p_i$, $msg.Ne \leftarrow Ne_i$, $msg.iee \leftarrow ie_i$, $msg.dest \leftarrow p_j$
11: Envoyer ($msg$, $VecteurDeDependances_i$) à $p_j$
12: Sauvegarder $msg$ en mémoire
13:
14: **Envoi d'un message** $msg$ **vers le monde extérieur**
15: **si** $\forall k \in \{0, ..., n\}, VecteurDeDependances_i[k] = \perp$ **alors**
16: Envoyer $msg$
17: **sinon**
18: Ajouter ($msg$, $VecteurDeDependances_i$) à $MessageExterieur_i$
19: **fin si**
20:
21: **Sur réception d'un message (**$msg$**,** $VecteurDeDependances_{msg}$**) venant du processus** $p_j$
22: $ie_i \leftarrow ie_i + 1$
23: Délivrer $msg$ à l'application
24: $VecteurDeDependances_i[i] \leftarrow ie_i$
25: **pour tous** $k$ **tels que** $VecteurDeDependances_{msg}[k] \neq \perp$ **faire**
26: **si** $VecteurDeDependances_i[k] \prec VecteurDeDependances_{msg}[k] \wedge VecteurStable_i[k] \prec VecteurDeDependances_{msg}[k]$ **alors**
27: $VecteurDeDependances_i[k] \leftarrow VecteurDeDependances_{msg}[k]$
28: **fin si**
29: **fin pour**
30: Initialiser $det_{msg}$ // $det_{msg}.Nr \leftarrow ie_i$
31: Sauvegarder $det_{msg}$ sur support stable // *asynchrone*
32:
33: **Sur réception de** $VecteurStable$
34: **pour tous** $k$ **tels que** $VecteurDeDependances[k] \prec VecteurStable[k]$ **faire**
35: $VecteurDeDependances[k] \leftarrow \perp$
36: **fin pour**
37: **pour tous** ($msg$, $VecteurDeDependances_{msg}$) $\in MessageExterieur_i$ **faire**
38: **pour tous** $k$ **tels que** $VecteurDeDependances_{msg}[k] \prec VecteurStable[k]$ **faire**
39: $VecteurDeDependances_{msg}[k] \leftarrow \perp$
40: **fin pour**
41: **si** $\forall k \in \{0, ..., n\}, VecteurDeDependances_{msg}[k] = \perp$ **alors**
42: Enlever ($msg$, $VecteurDeDependances_{msg}$) de $MessageExterieur_i$
43: Envoyer $msg$
44: **fin si**
45: **fin pour**
46: $VecteurStable_i \leftarrow VecteurStable$

FIG. 5.13 – Protocole O2P en l'absence de défaillance

est stable, cela signifie que tous les intervalles d'exécution du processus $p_i$ précédents $v$ sont eux aussi stables.

**Envoyer un message vers le monde extérieur** Un message ne peut être envoyé vers le monde extérieur que lorsque l'intervalle d'exécution d'émission est valide, c'est-à-dire que toutes les entrées du vecteur de dépendances associé à l'intervalle d'exécution valent $\perp$ (ligne 15). Dans le cas où l'intervalle d'exécution d'émission n'est pas valide, le message est stocké dans une liste nommée $MessageExterieur$ avec le vecteur de dépendances qui lui est associé (ligne 18). Lors de la réception d'un nouveau $VecteurStable$, les vecteurs de dépendances associés aux messages contenus dans la liste $MessageExterieur$ sont mis à jour et les messages valides sont envoyés (ligne 37-45). Ainsi les messages à envoyer vers le monde extérieur sont envoyés au plus tôt.

**Réduction de la quantité de données à attacher sur les messages** L'objectif du protocole O2P est de réduire au maximum la quantité de données à attacher sur chaque message de l'application. Dans un souci de lisibilité de l'algorithme, nous avons décrit qu'un processus attachait son vecteur de dépendance complet au message qu'il envoyait. En réalité, il n'est pas nécessaire d'attacher le vecteur complet à chaque message. *Seules les entrées du vecteur de dépendances non vides qui ont changé depuis le dernier envoi au même processus sont à attacher sur le message.* Ainsi si les déterminants sont sauvegardés assez rapidement, on se trouve dans la situation illustrée par la figure 5.9 : il n'y a aucune donnée à attacher sur les messages applicatifs. Pour mettre en œuvre ce dispositif, nous adaptons l'algorithme proposé par Singhal et Kshemkalyani [172]. Cette algorithme est décrit par la figure 5.14. La fonction permettant de calculer la liste des dépendances à attacher sur un message de l'application doit être appelée à la ligne 11 de la figure 5.13.

```
 1: Initialisation des variables
 2:     DernierMAJ_i ← [⊥, ..., ⊥]
 3:     DernierEnvoi_i ← [⊥, ..., ⊥]
 4:
 5: Sur mise à jour de VecteurDeDependances_i[k]
 6:     DernierMAJ_i[k] ← ie_i
 7:
 8: Calcul de la liste des données à attacher sur le message msg envoyé au processus p_j
 9:     ListeDeDependances_msg ← ⊥
10:     pour tous k tels que VecteurDeDependances_i[k] ≠ ⊥ faire
11:         si DernierEnvoi_i[j] < DernierMAJ_i[k] alors
12:             Ajouter VecteurDeDependances_i[k] à ListeDeDependances_msg
13:         fin si
14:     fin pour
15:     DernierEnvoi_i[j] ← ie_i
```

FIG. 5.14 – Calcul des dépendances à attacher sur un message

**Sauvegarde de points de reprise** Pour améliorer les performances au redémarrage et réduire les quantités d'informations sauvegardées aussi bien sur support stable pour les déterminants que dans la mémoire volatile des émetteurs pour le contenu des messages, des points de reprise sont utilisés.

Un point de reprise d'un processus peut être sauvegardé sans aucune coordination avec les autres processus de l'application. Il doit contenir en plus de l'état du processus, le contenu des messages sauvegardés en mémoire dans le cadre de l'enregistrement de

messages fondé sur l'émetteur au cas où un processus ait à nouveau besoin d'un de ces messages après un retour arrière.

Un point de reprise n'est valide que lorsque l'intervalle d'exécution dans lequel il a été sauvegardé devient valide (La validation des points de reprise fonctionne selon la même méthode que décrite précédemment pour les messages envoyés vers le monde extérieur). À partir de ce moment, le processus n'aura jamais de retour arrière à effectuer dans un état précédent ce point de reprise. Quand un point de reprise devient valide, les points de reprise précédents de ce même processus peuvent donc être supprimés tout comme les déterminants de messages ayant été reçus par ce processus avant ce dernier point de reprise valide. Enfin, quand un point de reprise d'un processus $p_i$ devient valide, l'information est envoyée à tous les processus de l'application pour qu'ils puissent supprimer de leur mémoire volatile le contenu des messages reçus par $p_i$ avant ce point de reprise.

L'utilisation de points de reprise permet d'améliorer les performances au redémarrage car au lieu de redémarrer depuis zéro et rejouer tous les messages lors d'un retour arrière, un processus peut redémarrer depuis son dernier point de reprise valide et ne rejouer que les messages reçus après ce point de reprise.

### 5.2.4 Prise en charge des défaillances

Dans ce paragraphe, nous décrivons la gestion des défaillances de processus par le protocole O2P. En cas de défaillance de processus de l'application, un protocole à enregistrement de messages optimiste doit tout d'abord détecter les processus orphelins et calculer l'état global cohérent maximum. Il effectue ensuite les retours arrières nécessaires et rejoue les messages jusqu'à rétablir l'application dans cet état global cohérent maximum. Nous présentons comment ceci est réalisé par O2P. De plus, nous prouvons que O2P rétablit toujours l'application dans un état global cohérent et que cet état est l'état global cohérent maximum quelque soit le nombre de processus défaillants.

**Définition 5.5 (État global cohérent maximum)** *L'état global cohérent maximum est parmi l'ensemble des états globaux cohérents qui peuvent être rétablis après une défaillance, celui qui correspond à l'état le plus avancé dans l'exécution de l'application.*

Nous supposons ici qu'un service tel que XtreemGCP se charge de redémarrer les processus défaillants.

Au cours de la description du protocole de recouvrement, nous allons être amenés à introduire de nouvelles structures de données. La figure 5.16 présente une version mise à jour de l'algorithme 5.13 prenant en compte ces nouvelles structures de données. Y sont grisées les modifications apportées par rapport à la version précédente de l'algorithme. Dans cette nouvelle version, nous avons supprimé la partie relative à la gestion des messages à envoyer vers le monde extérieur pour une meilleure lisibilité. Cette partie est de toute façon assez simple : lorsqu'un processus apprend un retour arrière, il teste sa liste de messages à envoyer vers le monde extérieur pour en supprimer les messages orphelins.

La première étape après une défaillance est de déterminer si certains déterminants ont été perdus et ont créé des processus orphelins. En cas de défaillance d'un seul processus, détecter les processus orphelins est simple car tous les processus ont leur vecteur de dépendances à jour excepté le processus fautif. Après avoir appris la défaillance du processus, un processus non fautif peut déterminer localement à partir des informations contenues dans son vecteur de dépendances si il est un processus orphelin.

Cependant quand plusieurs processus subissent une défaillance, la tâche devient plus complexe car les processus fautifs perdent les informations sur leurs dépendances lors de

la défaillance. Or les processus fautifs peuvent avoir des dépendances entre eux : l'état stable maximum d'un processus fautif peut être un état orphelin si il dépend d'un autre processus fautif.

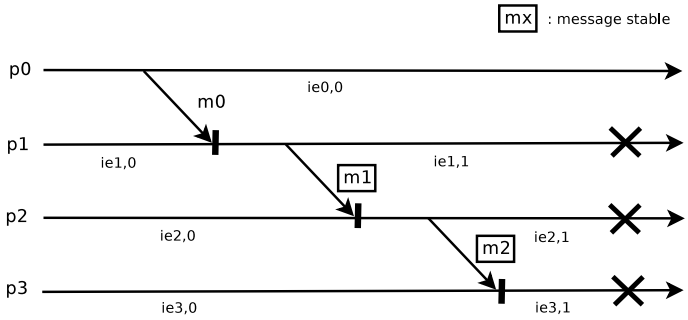



FIG. 5.15 – Exemple d'exécution avec plusieurs défaillances de processus

La figure 5.15 illustre le problème. Sur cette figure, les messages dont les déterminants ont été sauvegardés sur support stable, simplement appelés messages stables, sont encadrés. En se fondant sur les informations disponibles sur le support de stockage stable, le processus *p3* peut estimer qu'il est capable de restaurer son intervalle d'exécution *ie3,1*. Cependant cet intervalle d'exécution ne doit en réalité pas être restauré car il dépend causalement de l'intervalle d'exécution *ie1,1* qui lui ne peut pas être restauré, puisque le déterminant du message *m0* a été perdu lors de la défaillance. Ce simple exemple met en évidence le besoin d'un protocole entre les processus de l'application pour trouver l'état global cohérent maximum.

**Calcul centralisé de l'état global cohérent** Johnson et Zwaenepoel [106] ont décrit un algorithme centralisé permettant de calculer la ligne de recouvrement pendant l'exécution normale de l'application, en prenant en compte les déterminants sauvegardés sur support stable. Ceci implique qu'une entité centralisée doit connaître tous les déterminants sauvegardés sur support stable par l'ensemble des processus de l'application. Nous ne voulons pas nous fonder sur une solution centralisée car ce type de solution limite le passage à l'échelle et peut introduire un point unique de vulnérabilité aux défaillances.

#### 5.2.4.1 Redéfinition de l'ordre partiel entre les intervalles d'exécution

Quand un processus effectue un retour arrière, il faut être capable de différencier les intervalles d'exécution annulés par le retour arrière des intervalles d'exécution valides suivant ce retour arrière. C'est pourquoi un numéro d'incarnation doit être utilisé. Ce numéro est incrémenté à chaque retour arrière d'un processus et doit être inscrit sur support stable. Un intervalle d'exécution est alors identifié de manière unique par une paire composée du numéro d'incarnation du processus et de l'indice de l'intervalle d'exécution. La figure 5.17 montre comment les numéros d'incarnation permettent de différencier les intervalles d'exécution. Sur cet exemple, le numéro d'incarnation permet par exemple de différencier l'intervalle d'exécution 4 annulé par le retour arrière du nouvel intervalle d'exécution 4 créé après le retour arrière. Un message envoyé lors de l'intervalle d'exécution *ie0,(0,4)* est un message orphelin alors qu'un message envoyé lors de l'intervalle d'exécution *ie0,(1,4)* ne l'est pas.

1: **Initialisation des variables du processus** $p_i$
2: $VecteurDeDependances_i \leftarrow [\perp, ..., \perp]$ // *Vecteur de dépendances*
3: $ListeDeDependances_i \leftarrow \perp$
4: $VecteurStable_i \leftarrow [\perp, ..., \perp]$ // *Vecteur des états stables maximums connus*
5: $MessageExterieur_i \leftarrow \perp$ // *Liste de messages à envoyer vers le monde extérieur*
6: $Ni_i \leftarrow 0$ // *Numéro d'incarnation*
7: $id_i \leftarrow 0$ // *Indice de l'intervalle d'exécution*
8: $ie_i$ est défini comme $(Ni_i, id_i)$ // *Intervalle d'exécution*
9: $Ne_i \leftarrow 0$ // *Numéro d'émission*
10: $Etat_i \leftarrow normal$
11:
12: **Envoi d'un message** $msg$ **au processus** $p_j$
13: $Ne_i \leftarrow Ne_i + 1$
14: Initialiser $msg$ // $msg.source \leftarrow p_i$, $msg.Ne \leftarrow Ne_i$, $msg.iee \leftarrow ie_i$, $msg.dest \leftarrow p_j$
15: Envoyer $(msg, VecteurDeDependances_i)$ à $p_j$
16: Sauvegarder $msg$ en mémoire
17:
18: **Sur réception d'un message (**$msg$**,** $VecteurDeDependances_{msg}$**) venant du processus** $p_j$
19: **si** $Etat_i \neq normal$ **alors**
20: Repousser le traitement de $(msg, VecteurDeDependances_{msg})$
21: **fin si**
22: **si** $\exists k$ **tel que** $\neg Vivant_i(VecteurDeDependances_{msg}[k])$ **alors**
23: Éliminer $(msg, VecteurDeDependances_{msg})$
24: **sinon**
25: $id_i \leftarrow id_i + 1$
26: Délivrer $msg$ à l'application
27: $VecteurDeDependances_i[i] \leftarrow ie_i$
28: **pour tous** $k$ **tels que** $VecteurDeDependances_{msg}[k] \neq \perp$ **faire**
29: **si** $VecteurDeDependances_i[k] \prec VecteurDeDependances_{msg}[k] \wedge VecteurStable_i[k] \prec VecteurDeDependances_{msg}[k]$ **alors**
30: $VecteurDeDependances_i[k] \leftarrow VecteurDeDependances_{msg}[k]$
31: Ajouter $(k, VecteurDeDependances_{msg}[k], ie_i)$ à $ListeDeDependances_i$
32: **fin si**
33: **fin pour**
34: Initialiser $det_{msg}$ // $det_{msg}.Nr \leftarrow ie_i$
35: Sauvegarder $det_{msg}$ sur support stable // *asynchrone*
36: **fin si**
37:
38: **Sur réception de** $VecteurStable$
39: **pour tous** $k$ **tels que** $VecteurDeDependances[k] \prec VecteurStable[k]$ **faire**
40: $VecteurDeDependances[k] \leftarrow \perp$
41: **fin pour**
42: **pour tous** $(k, ie_k, ie_i) \in ListeDeDependances_i$ **tels que** $VecteurStable[k] \geq ie_k$ **faire**
43: Enlever $(k, ie_k, ie_i)$ de $ListeDeDependances_i$
44: **fin pour**
45: $VecteurStable_i \leftarrow VecteurStable$

FIG. 5.16 – Protocole O2P prenant en compte les défaillances

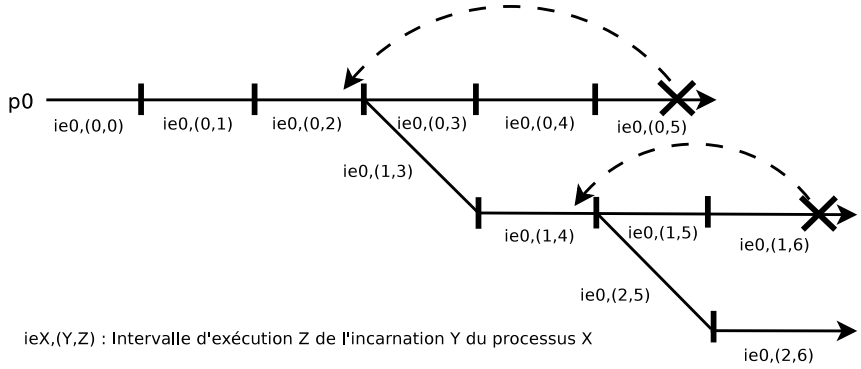



FIG. 5.17 – Identification des intervalles d'exécution d'un processus

Pour comparer des intervalles d'exécution, il faut donc prendre en compte leur numéro d'incarnation et leur indice. On note $Ni$, le numéro d'incarnation d'un intervalle d'exécution, et $id$, l'indice d'un intervalle d'exécution. La relation d'ordre sur les intervalles d'exécution d'un même processus est alors la suivante :

$$ie_a \prec ie_b \Leftrightarrow (ie_a.Ni < ie_b.Ni) \vee (ie_a.Ni = ie_b.Ni \wedge ie_a.id < ie_b.id)$$

Cette comparaison n'est cependant valide que pour les intervalles d'exécution appartenant à l'exécution du processus, c'est-à-dire les intervalles d'exécution n'ayant pas été annulés par un retour arrière. Par exemple, sur la figure 5.17, les intervalles d'exécution *ie0,(0,5)* et *ie0,(1,4)* ne sont pas comparables.

Pour déterminer quels intervalles d'exécution appartiennent à l'exécution d'un processus, chaque processus doit conserver un historique des retours arrière des autres processus. Cet historique contient l'indice du dernier intervalle d'exécution vivant, c'est-à-dire non annulé par un retour arrière, pour chaque incarnation d'un processus. Il est sauvegardé sur support stable. La figure 5.18 décrit l'historique du processus *p0* correspondant à l'exécution décrite sur la figure 5.17.

Historique du processus p0

[ie0,(0,2)]
[ie0,(1,4)]

FIG. 5.18 – Historique d'un processus

On définit le prédicat $Vivant_j(ieX,(Y,Z))$ sur le processus $p_j$ qui est vrai si l'intervalle d'exécution $ieX,(Y,Z)$ appartient à l'exécution du processus $p_x$ d'après les informations contenues dans l'historique de $p_x$ tenu à jour par $p_j$.

$$Vivant_j(ieX,(Y,Z)) = vrai \;\Leftrightarrow\; \nexists\, ieX,(Y,Z') \;\in\; Historique_j^x \text{ tel que } Z' < Z$$

Un processus $p_j$ dépendant d'un intervalle d'exécution $ie_k$, avec $Vivant_j(ie_k) = faux$, est un processus orphelin puisqu'il dépend d'un intervalle d'exécution qui a été annulé lors d'un retour arrière. Sur l'exemple décrit par les figures 5.17 et 5.18, considérons que

l'historique présenté est celui que possède un processus *p1* sur l'exécution de *p0*. Si l'état courant de *p1* dépend de l'intervalle d'exécution *ie0,(1,5)*, *p1* est un processus orphelin. En effet, $Vivant_1$*(ie0,(1,5))* est $faux$ car *ie0,(1,4)* appartient à l'historique de *p0* tenu à jour par *p1*.

#### 5.2.4.2 Redémarrage d'un processus fautif

L'algorithme utilisé au redémarrage d'un processus fautif est présenté par la figure 5.19. Après une défaillance, un processus fautif doit commencer par récupérer les informations sauvegardées sur support stable (ligne 3-5), c'est-à-dire le numéro d'incarnation avant la défaillance, son historique des retours arrière et la liste des déterminants sauvegardés avant la défaillance. Il peut ainsi évaluer l'indice de l'intervalle d'exécution maximum qu'il peut rejouer. Cet indice correspond en fait au nombre de déterminants sauvegardés, puisque les déterminants sont sauvegardés dans l'ordre. Il annonce cet intervalle d'exécution par un message $REDEMARRER$ envoyé à tous les processus de l'application (ligne 6). Enfin cette valeur est sauvegardée dans le vecteur, nommé $LdR$ pour ligne de recouvrement, qui va servir au calcul de l'état global cohérent maximum (ligne 7).

1: **Redémarrage du processus $p_i$ après une défaillance**
2: $Etat_i \leftarrow redemarrage$
3: $Ni_i \leftarrow Ni_{stable}$ // $Ni_{stable}$ *est le numéro d'incarnation inscrit sur support stable*
4: $Historique_i \leftarrow Historique_{stable}$
5: Obtenir $Journal_i$ sur le support stable // $Journal_i$ *est la liste des déterminants sauvegardés par* $p_i$
6: Envoyer $REDEMARRER(p_i, Ni_i, |Journal_i|)$ à tous les autres processus
7: $LdR_i[i] \leftarrow (Ni_i, |Journal_i|)$
8: Démarrer $TrouverEtatGlobalCoherentMaximum()$

FIG. 5.19 – Redémarrage après une défaillance

#### 5.2.4.3 Détecter les processus orphelins

Après la défaillance d'un ou plusieurs processus de l'application, les processus non fautifs doivent déterminer s'ils sont devenus des processus orphelins. Lorsqu'un processus non fautif reçoit une annonce de redémarrage, il peut utiliser son vecteur de dépendances pour savoir s'il est orphelin. Si l'intervalle d'exécution maximum que peut restaurer le processus $p_j$, annonçant son redémarrage, après une faute précède causalement le dernier intervalle d'exécution de $p_j$ dont l'état courant du processus $p_i$ dépend, alors $p_i$ est un processus orphelin (ligne 20 de la figure 5.20).

Une fois qu'un processus a déterminé qu'il est orphelin, il doit trouver son dernier intervalle d'exécution non orphelin, *i.e.* son intervalle d'exécution valide maximum ($ieVM$). Pour cela les informations contenues dans son vecteur de dépendance ne sont pas suffisantes. C'est pourquoi nous introduisons une nouvelle structure de données, la liste de dépendances ($ListeDeDependances_i$), qui contient l'ensemble des dépendances d'un processus $p_i$ et associe à chacune d'entre elles le premier intervalle d'exécution de $p_i$ ayant cette dépendance (ligne 31 de la figure 5.16). Cette liste de dépendances permet de retrouver le dernier intervalle d'exécution de $p_i$ qui ne devient pas orphelin en conséquence d'un retour arrière (ligne 21-23 de la figure 5.20).

**Message orphelin** Un message orphelin est un message qui dépend d'intervalles d'exécution qui ne peuvent pas être rejoués après une défaillance. Si un tel message est

reçu, il va créer un processus orphelin. Il faut donc éliminer ces messages. Pour déterminer si un nouveau message reçu est un message orphelin, nous utilisons l'historique des retours arrière. Si un intervalle d'exécution dont dépend le message n'est pas un intervalle d'exécution vivant d'après les informations contenues dans l'historique des retours arrière, alors le message est orphelin et ne doit pas être reçu par le processus applicatif (ligne 22-23 de la figure 5.16).

```
 1: Tâche TrouverIntervalleExecutionValideMaximum()
 2:    pour tous  k tels que VecteurDeDependances_i[k] ≠ ⊥ faire
 3:       Envoyer DEMANDE_STABLE(VecteurDeDependances_i[k]) au processus p_k
 4:    fin pour
 5:    Attendre que ∀k ∈ {0, ..., n}, VecteurDeDependances_i[k] = ⊥
 6:    Envoyer VALIDER(ieVM_i) à tous p_k ∈ ListePF_i
 7:    si ieVM_i ≺ ie_i alors
 8:       Démarrer RetourArriere(ieVM_i)
 9:    fin si
10:
11: Sur réception de REDEMARRER(Ni_j, id_j) venant du processus p_j
12:    ListePF_i ← ListePF_i ∪ id_j
13:    Ajouter (Ni_j, id_j) à Historique_i
14:    Sauvegarder (Ni_j, id_j) sur support stable
15:    si TrouverIntervalleExecutionValideMaximum() n'est pas en cours d'exécution alors
16:       ieVM_i ← ie_i    // ieVM_i est l'intervalle d'exécution valide maximum du processus p_i
17:       Etat_i ← recouvrement
18:       Démarrer TrouverIntervalleExecutionValideMaximum()
19:    fin si
20:    si (Ni_j, id_j) ≺ VecteurDeDependances_i[j] alors
21:       Soit ListeOrphelin_i l'ensemble des (j, ie_j, ie_i) ∈ ListeDeDependances_i tels que (Ni_j, id_j) ≺
          ie_j
22:       Soit ie_min la valeur minimale de ie_i ∀(j, ie_j, ie_i) ∈ ListeOrphelin_i
23:       ieVM_i ← minimum(ieVM_i, (ie_min.Ni, ie_min.id − 1))
24:       VecteurDeDependances_i[j] ← ⊥
25:    fin si
26:
27: Sur réception de DEMANDE_STABLE(Ni, id) venant du processus p_j
28:    Attendre que (Ni, id) ≺ VecteurStable_i[i]
29:    Envoyer REPONSE_STABLE(VecteurStable_i[i]) au processus p_j
30:
31: Sur réception de REPONSE_STABLE(Ni_j, id_j) venant du processus p_j
32:    si VecteurDeDependances_i[j] ≺ (Ni_j, id_j) alors
33:       VecteurDeDependances_i[j] ← ⊥
34:    fin si
```

FIG. 5.20 – Recouvrement pour les processus non fautifs

#### 5.2.4.4 Trouver l'état global cohérent maximum

En cas de défaillance d'un ou plusieurs processus, l'objectif est de rétablir l'application dans l'état global cohérent maximum. Pour cela, chaque processus de l'application va chercher à trouver son intervalle d'exécution valide maximum. Durant cette phase, appelée phase de recouvrement, les processus arrêtent de délivrer des messages applicatifs. En effet, il est inutile de continuer à délivrer des messages car si le processus est orphelin, les nouveaux intervalles d'exécution créés devront être annulés. C'est pourquoi nous avons introduit une variable $Etat_i$ définissant l'état du processus. Quand un processus est dans l'état *redemarrage* ou *recouvrement*, il ne peut pas recevoir de messages applicatifs (ligne 19-20 de la figure 5.16). La preuve que le protocole décrit permet de trouver l'état global cohérent maximum est fournie dans le paragraphe 5.2.4.5.

**Les processus non fautifs** Quand un processus non fautif reçoit un premier message annonçant le redémarrage d'un autre processus, il passe en phase de recouvrement et lance la tâche $TrouverIntervalleExecutionValideMaximum()$ pour trouver son intervalle d'exécution valide maximum (ligne 15-19 de la figure 5.20). Cette tâche attend que le vecteur de dépendances du processus soit vide pour déterminer si le processus doit faire un retour arrière. Ce vecteur se vide quand le processus reçoit les derniers acquittements du support stable (ligne 39-41 de la figure 5.16), quand il reçoit un message $REDEMARRER$ (ligne 24 de la figure 5.20), ou quand il reçoit un message $REPONSE\_STABLE$ (ligne 33 de la figure 5.20). Ces derniers messages sont reçus en réponse aux messages $DEMANDE\_STABLE$, qui sont envoyés par le processus non fautif pour obtenir les informations sur les intervalles d'exécution dont il dépend au début de la phase de recouvrement (ligne 3 de la figure 5.20). Seul les processus non fautifs répondent à ce type de messages car les processus fautifs envoient de toute façon un message $REDEMARRER$.

Pendant l'exécution de la tâche $TrouverIntervalleExecutionValideMaximum()$, le processus met à jour la valeur associé à son intervalle d'exécution valide maximum ($ieVM$) à chaque annonce de retour arrière comme décrit dans le paragraphe 5.2.4.3. Quand le vecteur de dépendance est vide, toutes les dépendances du processus ont été résolues. La valeur contenue dans $ieVM_i$ est alors l'intervalle d'exécution valide maximum du processus. La tâche $TrouverIntervalleExecutionValideMaximum()$ décide alors du retour arrière du processus si nécessaire (ligne 7-9 de la figure 5.20). Dans tous les cas, la valeur de cet intervalle d'exécution valide maximum est envoyé à tous les processus fautifs de l'application dans un message $VALIDER$ (ligne 6 de la figure 5.20).

**Les processus fautifs** Comme nous l'avons expliqué au travers de l'exemple de la figure 5.15, retrouver l'état valide maximum d'un processus fautif est plus complexe car celui-ci a perdu son vecteur de dépendances stocké en mémoire volatile lors de la défaillance. Certains protocoles existants résolvent ce problème en sauvegardant sur support stable, le vecteur de dépendance associé à chaque déterminant [53, 177]. Cette solution augmente considérablement la quantité d'information à sauvegarder sur support stable et risque donc de ralentir la sauvegarde des déterminants, compromettant le bon fonctionnement de O2P. C'est pourquoi nous n'avons pas choisi cette solution.

Pour déterminer l'état global cohérent maximum, les processus fautifs ont donc besoin d'une phase de communication supplémentaire décrite par la figure 5.21. Celle-ci est inspirée de l'algorithme proposé par Sistla et Welch [173].

Après une défaillance, un processus fautif annonce son redémarrage à tous les processus de l'application puis lance la tâche $TrouverEtatGlobalCoherentMaximum()$ (ligne 8 de la figure 5.19). Il attend ensuite une réponse de tous les processus de l'application pour former la ligne de recouvrement initiale, sauvegardée dans le vecteur $LdR_i$ (ligne 3 de la figure 5.21). Les processus non fautifs répondent aux processus fautifs par un message $VALIDER$ annonçant leur état valide maximum (ligne 6 de la figure 5.20). Ces processus n'ont pas besoin d'être inclus dans la suite des communications puisqu'ils peuvent déterminer directement leur état valide maximum à l'aide de leur vecteur de dépendances. La réponse des processus fautifs à un message $REDEMARRER$ est leur propre message $REDEMARRER$ (ligne 18 de la figure 5.21).

À partir de cette ligne de recouvrement initiale, les processus fautifs interagissent sur plusieurs itérations jusqu'à obtenir l'état global cohérent maximum (ligne 2-12 de la figure 5.21). Au début de chaque itération, la ligne de recouvrement est sauvegardée dans le vecteur $preLdR_i$. Chaque processus envoie aux autres processus fautifs, le nouvel intervalle

d'exécution qu'ils pensent être leur intervalle d'exécution valide maximum ($ieVM_i$) dans un message $PROPOSER$. Sur réception de ces messages, chaque processus met à jour sa ligne de recouvrement et l'estimation de son propre intervalle d'exécution stable maximum. Quand la ligne de recouvrement reste inchangée entre 2 itérations, c'est-à-dire que les valeurs de $preLdR_i$ et $LdR_i$ sont les mêmes, le processus sait que l'état global cohérent maximum a été trouvé. Il peut alors commencer son retour arrière.

```
1: Tâche TrouverEtatGlobalCoherentMaximum()
2:    tant que Etat_i = redemarrage faire
3:       Attendre que LdR_i soit complet
4:       si LdR_i ≠ preLdR_i alors
5:          preLdR_i ← LdR_i
6:          LdR_i[k] ← ⊥ pour tous p_k ∈ ListePF_i
7:          LdR_i[i] ← ieVM_i
8:          Envoyer PROPOSER(LdR_i[i]) à tous p_k ∈ ListePF_i
9:       sinon
10:         Démarrer RetourArriere(LdR_i[i])
11:      fin si
12:   fin tant que
13:
14: Sur réception de REDEMARRER(Ni_j, id_j) venant du processus p_j
15:   LdR_i[j] ← (Ni_j, id_j)
16:   Enlever de Journal_i les déterminants tels que det.source = p_j ∧ (Ni_j, id_j) ≺ det.iee
17:   ieVM_i ← (Ni_i, |Journal_i|)
18:   si p_j ∉ ListePF_i alors
19:      Envoyer REDEMARRER(LdR_i) à p_j
20:      ListePF_i ← ListePF_i ∪ id_j
21:   fin si
22:
23: Sur réception de VALIDER(Ni_j, id_j) venant du processus p_j
24:   LdR_i[j] ← (Ni_j, id_j)
25:   Enlever de Journal_i les déterminants tels que det.source = p_j ∧ (Ni_j, id_j) ≺ det.iee
26:   ieVM_i ← (Ni_i, |Journal_i|)
27:
28: Sur réception de PROPOSER(Ni_j, id_j) venant du processus p_j
29:   LdR_i[j] ← (Ni_j, id_j)
30:   Enlever de Journal_i les déterminants tels que det.source = p_j ∧ (Ni_j, id_j) ≺ det.iee
31:   ieVM_i ← (Ni_i, |Journal_i|)
```

FIG. 5.21 – Recouvrement pour les processus fautifs

Nous illustrons le fonctionnement de l'algorithme de recherche de l'état global cohérent maximum sur la figure 5.22. Cet exemple décrit la recherche de cet état global pour le scénario décrit par la figure 5.15. Pour chaque itération nous montrons : le message envoyé au début de l'itération, la valeur de la ligne de recouvrement après avoir reçu le message de chaque processus, et la nouvelle estimation de l'intervalle d'exécution valide maximum du processus après avoir pris en compte les messages reçus au cours de l'itération. Le processus *p0* étant non défaillant, il ne participe qu'a la première itération puisqu'à la fin de celle-ci il connaît son intervalle d'exécution valide maximum. Entre l'itération 3 et l'itération 4, la valeur de la ligne de recouvrement ne change pas, ce qui signifie que l'état global cohérent a été trouvé.

**Le retour arrière** Les actions à effectuer lors d'un retour arrière sont décrites sur la figure 5.23. Nous ne détaillons pas ici comment un processus peut redémarrer à partir d'un point de reprise ni comment il récupère les messages à rejouer qui doivent lui être fournis par les émetteurs de ces messages. Pour plus de détails, nous renvoyons le lecteur à l'article Johnson et Zwaenepoel traitant de la technique de sauvegarde de messages fondée sur l'émetteur [105]. Après avoir rejoué les messages, il est nécessaire de changer de numéro

Iteration 1

| | Message envoyé | LdR | ieVM |
|---|---|---|---|
| p0 | VALIDER(ie0,0) | | |
| p1 | REDEMARRER(ie1,0) | [0, 0, 1, 1] | 0 |
| p2 | REDEMARRER(ie2,1) | [0, 0, 1, 1] | 0 |
| p3 | REDEMARRER(ie3,1) | [0, 0, 1, 1] | 1 |

Iteration 2

| | Message envoyé | LdR | ieVM |
|---|---|---|---|
| p0 | | | 0 |
| p1 | PROPOSER(ie1,0) | [0, 0, 0, 1] | 0 |
| p2 | PROPOSER(ie2,0) | [0, 0, 0, 1] | 0 |
| p3 | PROPOSER(ie3,1) | [0, 0, 0, 1] | 0 |

Iteration 3

| | Message envoyé | LdR | ieVM |
|---|---|---|---|
| p0 | | | 0 |
| p1 | PROPOSER(ie1,0) | [0, 0, 0, 0] | 0 |
| p2 | PROPOSER(ie2,0) | [0, 0, 0, 0] | 0 |
| p3 | PROPOSER(ie3,0) | [0, 0, 0, 0] | 0 |

Iteration 4

| | Message envoyé | LdR | ieVM |
|---|---|---|---|
| p0 | | | 0 |
| p1 | PROPOSER(ie1,0) | [0, 0, 0, 0] | 0 |
| p2 | PROPOSER(ie2,0) | [0, 0, 0, 0] | 0 |
| p3 | PROPOSER(ie3,0) | [0, 0, 0, 0] | 0 |

FIG. 5.22 – Exemple d'exécution de l'algorithme de recherche de l'état global cohérent maximum

d'incarnation comme décrit dans le paragraphe 5.2.4.1 et de supprimer du support stable les déterminants associés aux intervalles d'exécution qui ont été annulés par le retour arrière.

1: **Tâche RetourArriere**($Ni, id$)
2: Rejouer les messages jusqu'à ce que $ie_i = (Ni, id)$
3: $Ni_i \leftarrow Ni_i + 1$
4: Sauvegarder $Ni_i$ sur support stable
5: Supprimer du support stable tous les $det_i$ tels que $ie_i \prec det_i$ // *déterminants invalidés par le retour arrière du processus* $p_i$
6: $Etat_i \leftarrow normal$
7: $ListePF_i \leftarrow \perp$

FIG. 5.23 – Retour arrière d'un processus

#### 5.2.4.5 Preuves

Dans ce paragraphe, nous prouvons qu'à la fin de la phase de recouvrement suivant une ou plusieurs défaillances, l'algorithme décrit précédemment rétablit l'application dans son état global cohérent maximum. On suppose ici que le nombre de défaillances est $r$. La notion de dépendance utilisée dans ce paragraphe fait référence à la relation d'ordre partiel entre les intervalles d'exécution des processus de l'application telle que définie dans le paragraphe 5.2.4.1.

**Lemme 5.1** *Le vecteur de dépendances d'un processus contient toutes les dépendances causales non stables de l'intervalle d'exécution courant du processus.*

**Preuve** À chaque fois qu'un message est délivré, le vecteur de dépendances attaché sur le message (ligne 15 de la figure 5.16), sert à mettre à jour les dépendances du processus destinataire (ligne 28-30 de la figure 5.16). Ainsi les dépendances transitives du processus sont tenues à jour dans le vecteur de dépendances. Lors d'une exécution sans défaillance, une dépendance peut être enlevée du vecteur de dépendance dans deux cas : (i) l'intervalle

d'exécution concerné devient stable (ligne 40 de la figure 5.16), (ii) un intervalle d'exécution plus récent du même processus doit être inclus dans le vecteur de dépendances (ligne 30 de la figure 5.16). Dans le cas (i), un intervalle stable ne peut être la cause d'un processus orphelin, donc l'information peut être retirée du vecteur de dépendance sans risque. Dans le cas (ii), le nouvel intervalle exécution dépend de l'ancien. Il n'y a donc pas de perte d'information quand le vecteur de dépendances est mis à jour. □

**Lemme 5.2** *Tous les processus orphelins vont ultimement effectuer un retour arrière.*

**Preuve** Un processus orphelin est un processus non défaillant. Il a donc son vecteur de dépendances et sa liste de dépendances à jour. Comme les canaux de communication sont fiables, tous les processus orphelins vont finir par recevoir le message $REDEMARRER$ envoyé par les processus fautifs (ligne 6 de la figure 5.19). Quand un processus non fautif reçoit un tel message, il évalue son vecteur de dépendances (ligne 20 de la figure 5.20). D'après le lemme 5.1, ce processus va inévitablement découvrir s'il est orphelin et décider d'un retour arrière. □

**Lemme 5.3** *Dans le cas d'un retour arrière, un processus orphelin est rétabli dans son intervalle d'exécution valide maximum.*

**Preuve** La liste de dépendances d'un processus $p_i$ contient pour chaque dépendance de $p_i$ par rapport à un intervalle d'exécution d'un autre processus de l'application, le premier intervalle d'exécution de $p_i$ ayant cette dépendance (ligne 31 de la figure 5.16). Après avoir détecter qu'il est orphelin, un processus cherche dans sa liste de dépendances l'ensemble des intervalles d'exécution qui doivent être annulés par un retour arrière (ligne 21 de la figure 5.20). Dans cet ensemble est sélectionné le plus ancien intervalle d'exécution associé à ces dépendances. L'estimation de l'intervalle d'exécution valide maximum ($ieVM$) est alors choisi comme étant le premier intervalle d'exécution du processus précédent cet intervalle d'exécution orphelin (ligne 22-23 de la figure 5.20).

Le vecteur de dépendance d'un processus non fautif finit par se vider lors de la phase de recouvrement. En effet, le processus arrête de délivrer de nouveaux messages quand il entre dans cette phase (ligne 19 de la figure 5.16). De plus, il finit par recevoir un message $REDEMARRER$ de tous les processus fautifs (ligne 6 de la figure 5.19) et un message $REPONSE\_STABLE$ de tous les processus non fautifs (ligne 29 de la figure 5.20) qui lui permettent de vider son vecteur de dépendances. Quand le vecteur de dépendance du processus est vide, la valeur stockée dans $ieVM$ est l'intervalle d'exécution valide maximum du processus, puisque cela signifie qu'on été traitées toutes les dépendances du processus d'après le lemme 5.1. □

**Lemme 5.4** *À la fin de la première itération de* $TrouverEtatGlobalCoherentMaximum()$*, l'intervalle d'exécution valide maximum d'au moins un processus fautif est connu.*

**Preuve** Soit $A = \{ie_1, ie_2, ..., ie_r\}$ l'ensemble des intervalles d'exécution valides maximum des processus fautifs. Il existe au moins un intervalle d'exécution $ie_x$ dans $A$ qui ne dépend d'aucun autre intervalle d'exécution de $A$ puisqu'il y a une relation d'ordre partiel entre les intervalles d'exécution de l'application. Supposons que l'intervalle d'exécution $ie_x$ du processus $p_x$ soit un tel intervalle d'exécution (Sur l'exemple des figures 5.15 et 5.22, *ie1,0* est cet intervalle d'exécution). Dans ce cas, $ie_x$ ne dépend que d'intervalles d'exécution valides maximum de processus non fautifs. La première itération

de $TrouverEtatGlobalCoherentMaximum()$ se termine quand l'estimation de ligne de recouvrement ($LdR$) des processus fautifs est complète, c'est-à-dire qu'ils ont reçus les messages $REDEMARRER$ (ligne 15 de la figure 5.21) de tous les processus fautifs et surtout les messages $VALIDER$ (ligne 24 de la figure 5.21) de tous les non fautifs. D'après les lemmes 5.2 et 5.3, ces messages $VALIDER$ contiennent l'état valide maximum des processus non fautifs. Donc à la fin de la première itération, $p_x$ a toutes les informations nécessaires pour trouver $ie_x$. □

**Lemme 5.5** *À chaque itération, au moins un nouvel intervalle d'exécution valide maximum est connu.*

**Preuve** Nous suivons le même raisonnement que précédemment. Soit $B$ l'ensemble des intervalles d'exécution valides maximum qui n'ont pas été trouvés au début de l'itération $k$. Il y a au moins un intervalle d'exécution $ie_y$ de l'ensemble $B$ qui ne dépend d'aucun autre intervalle d'exécution de $B$. L'intervalle d'exécution $ie_y$ est donc connu à la fin de l'itération $k$. □

**Lemme 5.6** *L'ensemble des intervalles d'exécution valides maximum forment l'état global cohérent maximum.*

**Preuve** Par définition, un état global cohérent est un état sans processus orphelin. L'ensemble des intervalles d'exécution valides maximum des processus est donc un état global cohérent. Nous en déduisons que c'est l'état global cohérent maximum. □

**Théorème 5.1** *Après la défaillance d'un ou plusieurs processus, O2P rétablit l'application dans son état global cohérent maximum.*

**Preuve** Nous considérons $r$ défaillances de processus. Nous déduisons des lemmes 5.2 et 5.3 qu'à la fin de la première itération, les processus non fautifs ont fait un retour arrière, si nécessaire, dans leur intervalle d'exécution valide maximum. Nous pouvons conclure des lemmes 5.4 et 5.5 que les intervalles d'exécution valides maximum des processus défaillants sont trouvés en au plus $r$ itérations. Le lemme 5.6 nous permet de conclure que O2P rétablit l'application dans son état global cohérent maximum. □

## 5.3 Gestion distribuée de la sauvegarde des messages

Dans le paragraphe 5.2, nous avons décrit le protocole O2P qui est fondé sur l'enregistrement de message optimiste actif. En sauvegardant au plus tôt les déterminants des messages applicatifs sur support stable, O2P limite le risque de création de processus orphelins et surtout tente de réduire la taille des informations à attacher sur les messages applicatifs pour limiter le surcoût induit sur le fonctionnement normal des applications.

Le comportement de O2P dépend clairement du temps nécessaire pour sauvegarder les déterminants sur support stable, comme nous l'avons expliqué dans le paragraphe 5.2.3.1. La mise en œuvre de la sauvegarde des déterminants sur support stable est donc un enjeu majeur.

Les travaux récents sur les protocoles à enregistrement de messages [33, 28] ont introduit la notion d'enregistreur d'événements. Un enregistreur d'événements est présenté comme un processus servant d'interface entre les processus applicatifs et le support de stockage stable : les processus de l'application envoient les données à sauvegarder à l'enregistreur d'événements qui les écrit sur le support de stockage stable avant d'envoyer un

acquittement au processus concerné. L'intérêt de l'enregistreur d'événements est qu'il est capable d'exécuter des actions spécifiques au protocole à enregistrement de message utilisé.

Les travaux précédemment cités utilisent un enregistreur d'événements centralisé. Dans ce paragraphe, nous commençons par évaluer O2P en utilisant le même type d'enregistreur d'événements que dans ces travaux. Nous montrons ainsi que O2P peut efficacement réduire la taille des données attachées sur les messages applicatifs par rapport aux protocoles optimistes existants. Mais nous montrons aussi qu'un enregistreur d'événements centralisé souffre d'un problème de passage à l'échelle.

Dans un deuxième temps, nous proposons un enregistreur d'événements distribué pour résoudre ce problème de passage à l'échelle. Cet enregistreur d'événements distribué utilise la mémoire volatile des nœuds sur lesquels s'exécute l'application pour mettre en œuvre le support de stockage tolérant aux fautes : copier chaque donnée dans la mémoire volatile de $f + 1$ nœuds permet de tolérer $f$ défaillances de nœuds. Pour diffuser l'information sur les nouvelles données qui ont été sauvegardées sur support stable, notre enregistreur d'événements distribué se fonde sur un simple algorithme épidémique [59]. Les évaluations montrent que l'enregistreur d'événements distribué assure le passage à l'échelle des protocoles à enregistrement de messages. De plus, elles montrent que l'algorithme épidémique permet de réduire efficacement la taille des données attachées sur les messages applicatifs par O2P.

Dans ce paragraphe nous commençons par présenter la méthode et le matériel employé pour nos évaluations. Nous décrivons ensuite la mise en œuvre de O2P dans la bibliothèque Open MPI. Puis nous présentons les résultats des évaluations menées en utilisant un enregistreur d'événements centralisé. Enfin nous présentons notre solution pour une gestion distribuée des déterminants à sauvegarder de façon stable, et présentons les résultats de l'évaluation de cette solution mettant en évidence le passage à l'échelle de la solution fournie par la combinaison de O2P avec cet enregistreur d'événements distribué.

### 5.3.1 Méthode d'évaluation

Dans ce paragraphe, nous décrivons le matériel et les applications employés pour nos évaluations. Les évaluations présentées dans ce paragraphe concernent les performances de O2P en fonctionnement normal. Nous avons menées nos évaluations avec la révision 22041 de Open MPI disponible sur le serveur *http ://svn.open-mpi.org/svn/ompi/trunk*.

Nous n'avons pu évaluer les performances en recouvrement à cause des limites actuelles de Open MPI. En effet, l'intérêt principal des protocoles à enregistrement de messages par rapport aux protocoles de sauvegarde de points de reprise coordonnés est d'éviter de redémarrer tous les processus en réponse à la défaillance d'un seul processus. Cependant, actuellement la défaillance d'un processus MPI entraîne l'arrêt de tous les processus de l'application MPI, ne permettant donc pas de mettre en évidence les propriétés d'un protocole à enregistrement de messages en recouvrement.

#### 5.3.1.1 Plate-forme d'expérimentation

Toutes nos expériences sont menées sur le site de Rennes de la grille expérimentale Grid'5000 [84]. Il offre 161 machines de trois types. Elles sont toutes équipées d'un lien Gigabit Ethernet et sont interconnectées par un Switch Cisco 6509. Les trois types de machines sont :

**Paradent :** 64 nœuds équipés de processeurs Intel Xeon L5420 à 2.5 Ghz (2 processeurs composés de 4 cœurs par nœud), et de 32 GB de mémoire.

**Paramount :** 33 nœuds équipés de processeurs Intel Xeon 5148 LV à 2.33 Ghz (2 processeurs composés de 2 cœurs par nœud), et de 8 GB de mémoire.

**Paraquad :** 64 nœuds équipés de processeurs Intel Xeon 5148 LV à 2.33 Ghz (2 processeurs composés de 2 cœurs par nœud), et de 4 GB de mémoire.

Dans la suite du paragraphe, nous ferons référence à ces nœuds directement par leur nom.

#### 5.3.1.2 Applications employées

Pour évaluer nos prototypes, nous utilisons deux types d'applications. Nous utilisons tout d'abord l'application NetPIPE [179] qui permet de mesurer la latence et la bande passante entre deux machines sur un test de type *Ping-Pong*. Puis nous utilisons les 6 applications du NAS Parallel Benchmark [16] (BT, CG, FT, LU, MG, et SP) qui sont des applications représentatives des applications du domaine du calcul scientifique pour comprendre le comportement de nos prototypes avec des applications réelles. Les applications BT et SP doivent être exécutées avec un nombre de processus qui soit un carré. Pour les autres applications, le nombre de processus doit être une puissance de 2. Nous avons présenté les caractéristiques de ces applications en terme de messages échangés dans le tableau 5.1. Dans nos expériences, nous utilisons les classes B et C de ces applications, les classes correspondant à la taille du problème résolu.

### 5.3.2 Mise en œuvre de O2P dans Open MPI

Nous avons mis en œuvre le protocole O2P dans la bibliothèque Open MPI. Dans ce paragraphe, nous donnons les détails importants relatifs à cette mise en œuvre. Tout d'abord, nous décrivons l'intégration du module O2P à Open MPI. Nous décrivons ensuite la solution que nous avons choisie pour attacher des données sur les messages envoyés par les processus de l'application MPI.

#### 5.3.2.1 Le module O2P

Open MPI est fondé sur une architecture modulaire permettant d'intégrer simplement de nouvelles contributions [79]. Pour chaque fonctionnalité importante est définie une interface appelée *composant*. Une interface est mise en œuvre par un module. Plusieurs modules peuvent être définis pour une interface. Les modules à utiliser sont choisis à l'exécution de l'application.

Dans Open MPI, un composant nommé *PML* (*Point-to-point Management Layer*) est chargé des émissions et réceptions de messages entre processus d'une application. Un module nommé *PML-V* (*Vampire PML*) permet de surcharger le module *PML* sélectionné pour exécuter du code spécifique à chaque appel de fonction. Cette solution offre la possibilité de mettre en œuvre un algorithme à enregistrement de messages puisque qu'elle permet d'exécuter des actions à chaque émission et réception de messages. O2P est mis en œuvre au sein du module *PML-V*.

#### 5.3.2.2 Attacher des données sur les messages de l'application

Plusieurs solutions sont envisageables pour attacher des données sur les messages d'une application MPI. Les deux principales sont : (i) créer un nouveau type de données (*datatype*) à partir des données contenues dans le message original et des données à attacher ; (ii) envoyer un deuxième message en plus de l'original contenant les données supplémentaires. Schulz et al. ont effectué une étude comparative de ces solutions [169].

Leur étude montre que les performances de ces solutions dépendent de l'application considérée et de la bibliothèque MPI. Les tests que nous avons effectués ont montré que la deuxième solution était plus efficace dans notre cas.

Comme nous l'avons vu dans le paragraphe 5.2.3.1, O2P doit permettre de réduire au maximum la quantité de données à attacher sur les messages de l'application. Dans le meilleur des cas, aucune donnée n'a à être attachée sur les messages applicatifs. Cependant dans ce cas, comme à chaque message applicatif est supposé être associé un deuxième message contenant les données supplémentaires, il faut tout de même envoyer un message vide comme deuxième message. Pour éviter cet envoi de message inutile, nous avons modifié le *PML* par défaut utilisé par Open MPI, nommé *ob1*. Nous avons utilisé un bit de l'entête du message applicatif pour avertir de l'arrivée d'un deuxième message contenant les données supplémentaires. Ainsi le deuxième message n'a besoin d'être envoyé que si il existe réellement des données à transmettre.

Un échange de messages dans MPI se fait dans le contexte d'un *communicator*. Un processus faisant partie d'un *communicator* peut envoyer un message à tout processus faisant partie du même *communicator*. Un message envoyé dans le contexte d'un *communicator* ne peut être reçu que dans le même contexte. Pour éviter de confondre un message contenant les données additionnelles attachées sur un message applicatif avec un autre message applicatif, nous créons un *communicator* dédié aux envois des messages additionnels contenant tous les processus de l'application. Les canaux de communication FIFO entre les processus MPI assurent que les messages additionnels associés à deux messages applicatifs différents ne seront pas intervertis.

#### 5.3.2.3 Enregistrement de messages fondé sur l'émetteur

Quand un message est envoyé, le contenu de ce message est copié dans un fichier projeté en mémoire. Pour cela sont utilisés les mécanismes de gestion des types de données de Open MPI, en charge de mettre en forme les données à envoyer, potentiellement stockées à différentes adresses mémoires, dans un format exploitable par le destinataire du message. Les données sont ensuite écrites de manière asynchrone sur disque. Cette solution nous a été proposée par Aurélien Bouteiller de *Innovative Computing Laboratory, University of Tennessee*.

### 5.3.3 Évaluation de O2P avec un enregistreur d'événements centralisé

L'enregistreur d'événements centralisé que nous utilisons pour nos évaluations met en œuvre l'algorithme décrit par la figure 5.12 : après avoir écrit un déterminant sur support stable, il envoie comme acquittement le vecteur mis à jour composé de l'état stable maximum de chaque processus de l'application.

Nous commençons par décrire la mise en œuvre de l'enregistreur d'événements dans Open MPI. Puis nous présentons les performances en terme de latence et de bande passante de O2P avec cet enregistreur d'événements. Enfin nous menons des expériences avec les applications du NAS Parallel Benchmark. Nous évaluons tout d'abord la taille des données attachées sur les messages applicatifs par O2P, puis l'impact de O2P sur les performances de l'application.

#### 5.3.3.1 Mise en œuvre de l'enregistreur d'événements centralisé dans Open MPI

Nous avons mis en œuvre l'enregistreur d'événements centralisé sous la forme d'un processus MPI. Ce processus MPI additionnel doit être démarré avant le début de l'application. Les processus de l'application se connectent à l'enregistreur d'événements au moment du premier déterminant à enregistrer.

Pour obtenir les meilleurs performances possibles, l'enregistreur d'événements sauvegarde les déterminants qui lui sont envoyés en mémoire volatile.

#### 5.3.3.2 Évaluation de la latence et de la bande passante

Nous évaluons la latence et la bande passante avec l'application NetPIPE sur 3 nœuds *paramount*. L'enregistreur d'événements est exécuté sur un des nœuds, les deux autres nœuds servant à l'exécution de NetPIPE. Les résultats sur présentés sur la figure 5.24. Ils comparent les performances de Open MPI avec les performances de Open MPI quand O2P est activé. Nous présentons aussi les résultats quand les mécanismes d'enregistrement du contenu des messages sur l'émetteur sont désactivés dans O2P (courbe *O2P (sans sender-based)*).

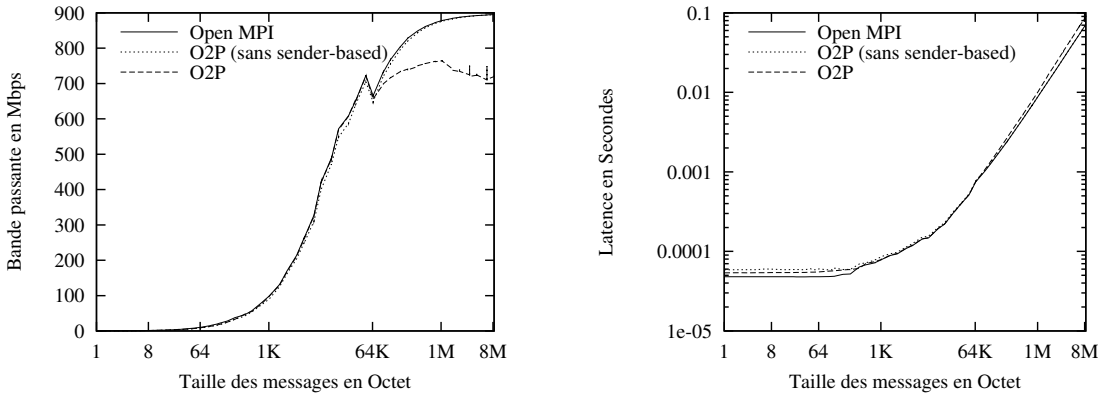



FIG. 5.24 – Bande passante et latence de O2P avec un enregistreur d'événements centralisé

Ces résultats montrent que les performances de O2P sont très proches des performances originales de Open MPI. C'est le résultat attendu car au pire lors de l'exécution d'une application composée de deux processus, 1 estampille[4] est attachée sur chaque message, ne pénalisant que très peu les performances de l'application. La comparaison des performances de O2P avec et sans enregistrement du contenu des messages sur l'émetteur montre que ce type de techniques peut avoir un impact sur les performances de l'application quand la taille des messages à enregistrer devient grande, *i.e.* supérieure à 64K octets dans notre cas. Cependant sauvegarder le contenu de ces messages sur l'enregistreur d'évènements serait aussi très coûteux.

#### 5.3.3.3 Taille des données attachées sur les messages

Pour mesurer le nombre d'estampilles attachées sur les messages applicatifs, nous menons une expérience sur 128 machines (44 *paradent*, 26 *paramount*, et 58 *paraquad*). Une

[4]Nous appelons *estampille* les données attachées sur un message applicatif pour représenter une dépendance.

machine *paradent* supplémentaire est utilisée pour exécuter l'enregistreur d'événements. Nous exécutons la classe B des applications du NAS Parallel Benchmark et faisons varier le nombre de processus de 8 à 256. Les résultats proposés sont des moyennes sur 5 exécutions de chaque application.

Nous comparons le nombre d'estampilles qu'attacherait un protocole optimiste n'essayant pas de réduire la taille des données attachées sur les messages avec les valeurs pour notre protocole optimiste actif. Pour une comparaison équitable, nous appliquons aussi au protocole optimiste *classique* l'algorithme de Singhal et Kshemkalyani [172], que nous avons présenté dans le paragraphe 5.2.3.3, qui permet une mise en œuvre optimisée des vecteurs d'horloge.

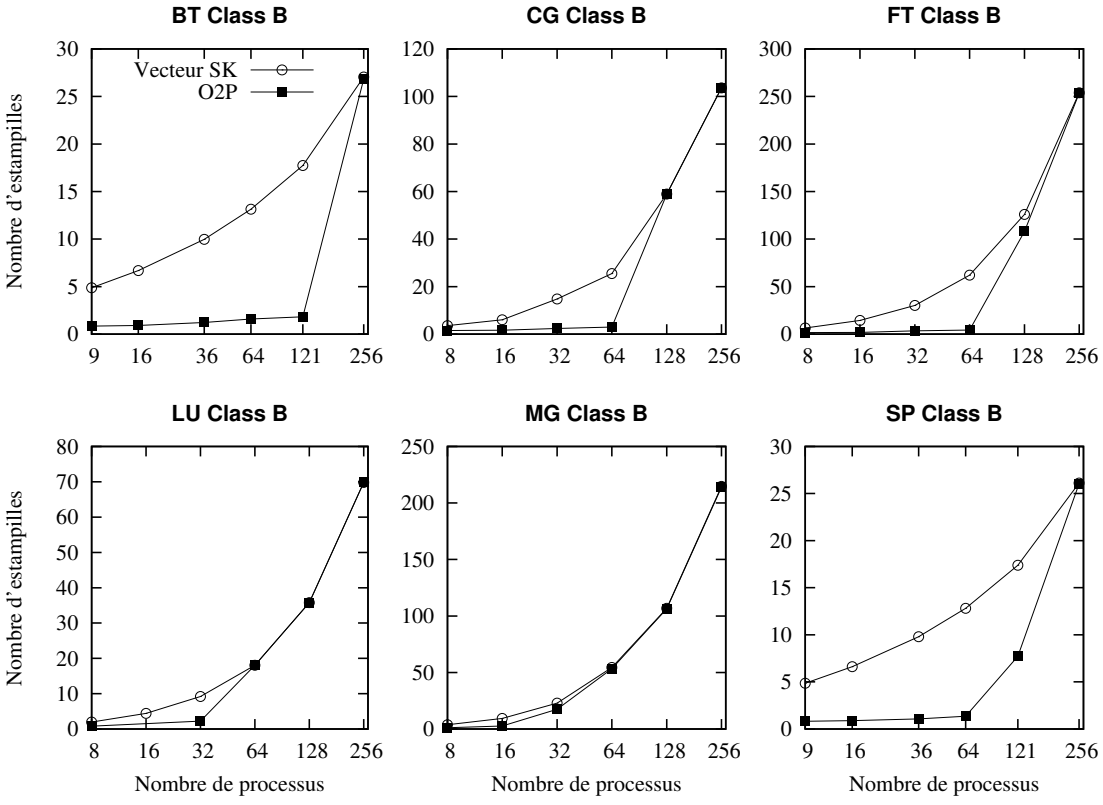



FIG. 5.25 – Quantité de données attachées sur les messages de l'application avec un enregistreur d'événements centralisé

Les résultats sont présentés sur la figure 5.25. Les courbes nommées *Vecteur SK* sont les résultats pour un protocole optimiste *classique* auquel est appliqué l'algorithme de Singhal et Kshemkalyani. Les courbes nommées *O2P* sont les résultats pour notre protocole.

Ces résultats montrent tout d'abord que O2P peut réduire efficacement la nombre d'estampilles attachées sur les messages applicatifs. Cependant ils mettent aussi en évidence les limites de l'enregistreur d'événements centralisé. Quand la fréquence des messages échangés par l'application est grande, comme dans le cas des applications MG et LU, ou quand le nombre de processus dans l'application devient trop grand, à 64 ou 128 processus selon l'application, il n'arrive plus à sauvegarder les déterminants assez rapidement pour permettre à O2P de fonctionner de façon optimale.

#### 5.3.3.4 Performance des applications

Pour mesurer les performances des applications en fonctionnement normal avec O2P, nous utilisons 128 machines (47 *paradent*, 27 *paramount*, et 54 *paraquad*). Une machine *paradent* supplémentaire est utilisée pour exécuter l'enregistreur d'événements. Nous exécutons la classe C des applications du NAS Parallel Benchmark et faisons varier le nombre de processus de 32 à 128. Les résultats proposés sont des moyennes sur 10 exécutions de chaque application.

Nous ne présentons pas ici les résultats pour les applications BT et SP car nous avons constaté des variations importantes dans les temps d'exécution de ces applications, rendant l'exploitation des résultats impossible. Nous avons constaté ces variations même lors des exécutions sans mécanismes de recouvrement arrière activés, mais n'avons pas réussi à trouver la cause de ces variations.

Dans cette expérience, nous comparons tout d'abord les performances de O2P avec celles d'un protocole à enregistrement de messages pessimiste. Notre mise en œuvre d'un protocole pessimiste est une modification du protocole O2P : un processus doit attendre que son vecteur de dépendances soit vide avant de pouvoir envoyer un message. Les résultats sont présentés sur la figure 5.26. Sur cette figure, nous représentons le surcoût induit par chaque protocole par rapport à l'exécution avec Open MPI sans tolérance aux fautes. Les valeurs représentées sont les temps d'exécution normalisés par rapport au temps d'exécution sans tolérance aux faute, *i.e.* la valeur associé au temps d'exécution sans tolérance aux fautes est de 1.

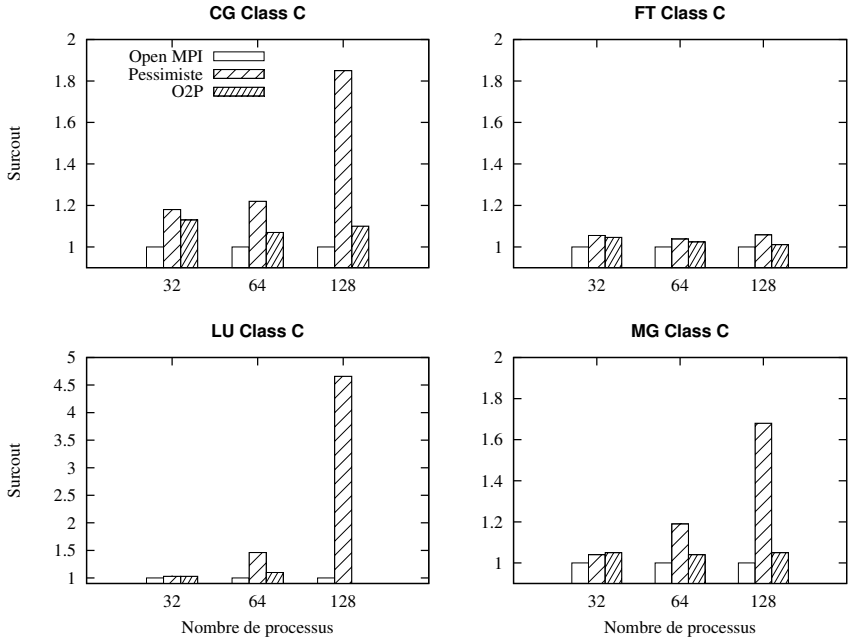



FIG. 5.26 – Performances de O2P et d'un protocole à enregistrement de messages pessimiste avec un enregistreur d'événements centralisé

Les résultats montrent tout d'abord comme attendu que O2P offre de bien meilleures performances en fonctionnement normal qu'un protocole pessimiste. FT est le seul test où les performances sont comparables. En effet, la fréquence des échanges de messages est faible pour cette application (cf. tableau 5.1). Ainsi, la plupart du temps les déterminants associés aux réceptions de messages ont le temps d'être sauvegardés avant l'émission du

message suivant. Le protocole pessimiste ne retarde alors pas les émissions de messages et ne pénalise donc pas les performances de l'application.

Comme avec un protocole pessimiste, un processus doit attendre les acquittements de l'enregistrement des déterminants pour les messages qu'il a reçu avant d'envoyer un message, la surcharge de l'enregistreur d'événements à un impact très important sur les performances du protocole. Ceci est remarquable avec CG. Nous avions constaté lors de l'expérience précédente que l'enregistreur d'événements était surchargé à partir de 128 processus. Or nous pouvons constater que pour cette taille d'application, le surcoût induit par le protocole pessimiste sur l'exécution de CG devient très important, le temps d'exécution de l'application étant presque multiplié par 2. Pour LU, la surcharge de l'enregistreur d'événements est encore plus importante, conduisant à un temps d'exécution multiplié par 4 pour 128 processus. Nous n'avons pas de résultat avec O2P pour ce test car avec O2P les processus continuent à envoyer de nouveaux déterminants à l'enregistreur d'événements même quand celui est déjà surchargé, conduisant à sa défaillance.

La figure 5.27 compare ensuite les performances de O2P avec les performance d'un protocole optimiste classique auquel serait appliqué l'algorithme de Singhal et Kshemkalyani. Les performances des deux protocoles sont équivalentes. En effet les tailles d'applications considérées ne sont pas suffisantes pour que la réduction du nombre d'estampilles attachées sur les messages, qui est au maximum égal au nombre de processus dans l'application, ait un impact significatif sur les performances.

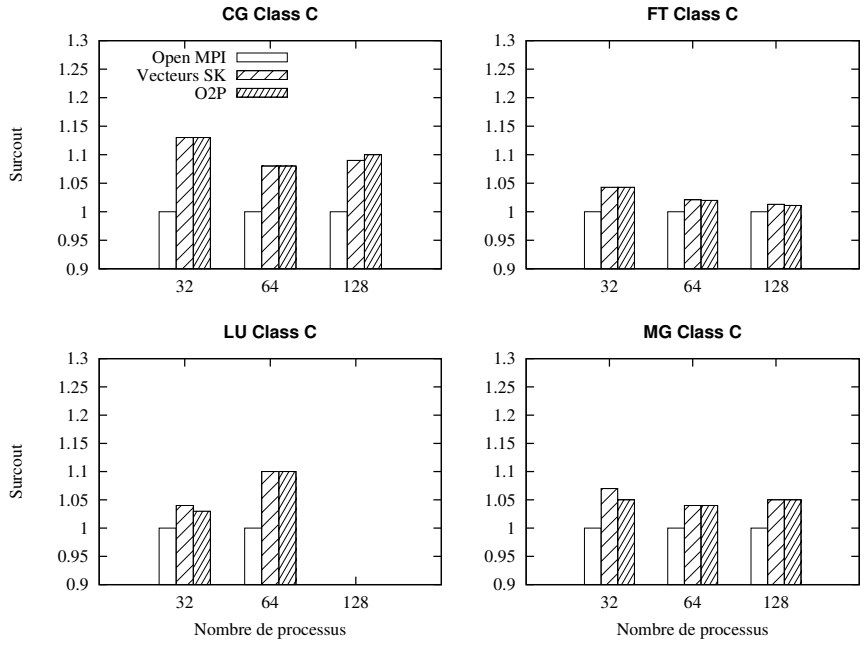



Fig. 5.27 – Performances de O2P et d'un protocole à enregistrement de messages optimiste classique avec un enregistreur d'événements centralisé

#### 5.3.3.5 Bilan

Ces expériences montrent tout d'abord les bonnes performances d'un protocole optimiste quand peu de données sont attachées sur les messages applicatifs. Ainsi lors des évaluations de la latence et de la bande passante sur un test de type *Ping-Pong*, les surcoûts constatés sont minimes. Le seul surcoût vient alors de l'enregistrement de messages fondé

sur l'émetteur pour des messages de grande taille. Ces expériences mettent ensuite en évidence la capacité de O2P à réduire le nombre d'estampilles attachées sur les messages applicatifs. Elles mettent surtout en avant les limites en terme de passage à l'échelle d'un enregistreur d'événements centralisé.

### 5.3.4 Évaluation de O2P avec un enregistreur d'événements distribué

Pour un meilleur passage à l'échelle nous proposons un enregistreur d'événements distribué. Dans ce paragraphe, nous commençons par décrire le fonctionnement de cet enregistreur d'événements et sa mise en œuvre dans Open MPI. Nous présentons ensuite les évaluations que nous avons menées sur Grid'5000.

#### 5.3.4.1 Description de l'enregistreur d'événements distribué

Pour avoir un stockage de données tolérant $f$ fautes, il faut dupliquer les données $f+1$ fois. C'est le principe que nous avons utilisé pour notre enregistreur d'événements distribué. Son fonctionnement est décrit par la figure 5.28. Nous n'incluons pas ici à nouveau tous les détails sur le fonctionnement de O2P que nous avons déjà décrit dans le paragraphe 5.2 mais seulement les détails permettant de comprendre le fonctionnement de l'enregistreur d'événements distribué.

1: **Initialisation des variables pour l'enregistreur d'événements** $eevt_q$ **situé sur le nœud** $q$
2: $ListeAcq_q \leftarrow \perp$
3: $VecteurStable_q \leftarrow [\perp, ..., \perp]$
4:
5: **Envoi d'un message** $msg$ **au processus** $p_j$ **par le processus** $p_i$ **situé sur le nœud** $q$
6: Calculer $ListeDeDependances_{msg}$ en comparant $VecteurDeDependances_i$ et $VecteurStable_q$
7: Envoyer ($msg$, $ListeDeDependances_{msg}$) à $p_j$
8:
9: **Sur réception d'un message (**$msg$**,** $ListeDeDependances_{msg}$**) par le processus** $p_i$ **situé sur le nœud** $q$
10: Délivrer $msg$ à l'application
11: Mettre à jour $VecteurDeDependances_i$ avec $ListeDeDependances_{msg}$
12: Initialiser $det_{msg}$
13: Envoyer ($det_{msg}$, $q$) à $f$ enregistreurs d'événements
14:
15: **Sur réception d'un déterminant (**$det_{msg}$**,** $q$**) par l'enregistreur d'événements** $eevt_r$ **situé sur le nœud** $r$
16: Sauvegarder $det_{msg}$ en mémoire volatile
17: Envoyer $acq_{msg}$ à $eevt_q$ // $acq_{msg}.ps \leftarrow det_{msg}.dest$, $acq_{msg}.ie \leftarrow det_{msg}.Nr$
18:
19: **Sur réception de** $acq_{msg}$ **par l'enregistreur d'événements** $eevt_q$ **situé sur le nœud** $q$
20: **si** $acq_{msg} \in ListeAcq_q$ **alors**
21: $ListeAcq_q[acq_{msg}] \leftarrow ListeAcq_q[acq_{msg}] + 1$
22: **sinon**
23: $ListeAcq_q[acq_{msg}] \leftarrow 1$
24: **fin si**
25: **si** $ListeAcq_q[acq_{msg}] = f$ **alors**
26: $VecteurStable_q[acq_{msg}.ps] \leftarrow acq_{msg}.ie$
27: Envoyer $VecteurStable_q$ à $g$ enregistreurs d'événements choisis aléatoirement
28: **fin si**
29:
30: **Sur réception de** $VecteurStable_q$ **par l'enregistreur d'événements** $eevt_r$ **situé sur le nœud** $r$
31: Mettre à jour $VecteurStable_r$ avec $VecteurStable_q$

FIG. 5.28 – Fonctionnement de l'enregistreur d'événements distribué

Un enregistreur d'événements est positionné sur chaque nœud où s'exécute un processus de l'application. Nous supposons que les nœuds ont un identifiant unique et que

ces identifiants sont totalement ordonnés. Chaque enregistreur d'événements tient à jour un $VecteurStable$ qui contient les états stables maximums qu'il connaît pour les processus de l'application. Pour gérer les acquittements, un enregistreur d'événements a une liste *ListeAcq* dans lequel il mémorise les différents acquittements qu'il a reçus. À chaque acquittement de la liste est associé le nombre de fois qu'il a été reçu.

Pour sauvegarder un déterminant de façon stable, un processus $p_i$ envoie celui-ci à $f$ enregistreurs d'événements (ligne 13). Ces $f$ enregistreurs d'événements sont choisis comme étant ceux situés sur les nœuds dont les identifiants sont les successeurs de l'identifiant du nœud hébergeant le processus $p_i$.

Quand un enregistreur d'événements reçoit un déterminant du processus $p_i$, il le sauvegarde en mémoire et envoie un acquittement à l'enregistreur d'événements du nœud hébergeant le processus $p_i$ (lignes 16-17). Quand un enregistreur d'événements a reçu les $f$ acquittements pour un déterminant, il sait que le déterminant est sauvegardé de façon stable et met donc à jour son $VecteurStable$ (lignes 25-28). Dans le cas de nœuds munis de plusieurs processeurs et/ou de processeurs multi-cœurs, il est possible que plusieurs processus de l'application soient exécutés sur le même nœud. Dans ce cas l'enregistreur d'événements de ce nœud traite les acquittements pour tous ces processus.

Un processus utilise le $VecteurStable$ fournit par l'enregistreur d'événements du nœud sur lequel il s'exécute pour connaître les nouveaux états stables des différents processus de l'application et calculer la liste de dépendances qu'il doit attacher sur les messages qu'il envoie (lignes 6-7).

Pour diffuser les informations sur les nouveaux état stables à l'ensemble des processus de l'application, un algorithme épidémique est utilisé. Après avoir mis son $VecteurStable$ à jour lorsqu'un déterminant devient stable, un enregistreur d'événements sélectionne aléatoirement $g$ enregistreurs d'événements auxquels il envoie son nouveau $VecteurStable$ (ligne 27). Ces enregistreurs d'événements utilisent cette information pour mettre à jour leur propre $VecteurStable$ (ligne 31). Nous appelons $g$ le *degré de diffusion* de notre algorithme.

#### 5.3.4.2 Mise en œuvre de l'enregistreur d'événements distribué dans Open MPI

Nous avons mis en œuvre l'enregistreur d'événements distribué sous la forme d'une application MPI. L'enregistreur d'événements doit être démarré avant l'application MPI de l'utilisateur. Le nombre de processus composant l'enregistreur d'événements est égal au nombre de nœuds sur lesquels doit être exécutée l'application de l'utilisateur : un processus de l'enregistreur d'événements s'exécute sur chacun de ces nœuds. Au démarrage de l'application de l'utilisateur, un appel à *MPI_COMM_CONNECT* permet de créer un *communicator* dans lequel l'ensemble des processus de l'application est connecté avec l'ensemble des processus de l'enregistreur d'événements. Tous ces processus peuvent ainsi s'envoyer des messages.

Le processus de l'enregistreur d'événements situé sur un nœud partage son $VecteurStable$ avec les processus de l'application situés sur ce nœud au travers d'un segment mémoire partagée System V.

#### 5.3.4.3 Évaluation de la latence et de la bande passante

Nous évaluons la latence et la bande passante avec l'application NetPIPE sur 2 nœuds *paramount*. Les résultats sur présentés sur la figure 5.24. Ils comparent les performances de Open MPI avec les performances de Open MPI quand O2P est activé. Nous présentons

aussi les résultats quand les mécanismes d'enregistrement du contenu des messages sur l'émetteur sont désactivés dans O2P (courbe *O2P (sans sender-based)*).

Dans cet expérience, chaque processus enregistre ces déterminants sur l'enregistreur d'événements situé sur le nœud hébergeant l'autre processus. Le degré de diffusion de l'enregistreur d'événements est fixé à 0 car l'objectif dans cette expérience n'est pas de réduire le nombre d'estampilles attachées sur les messages, ce nombre étant au maximum de 1.

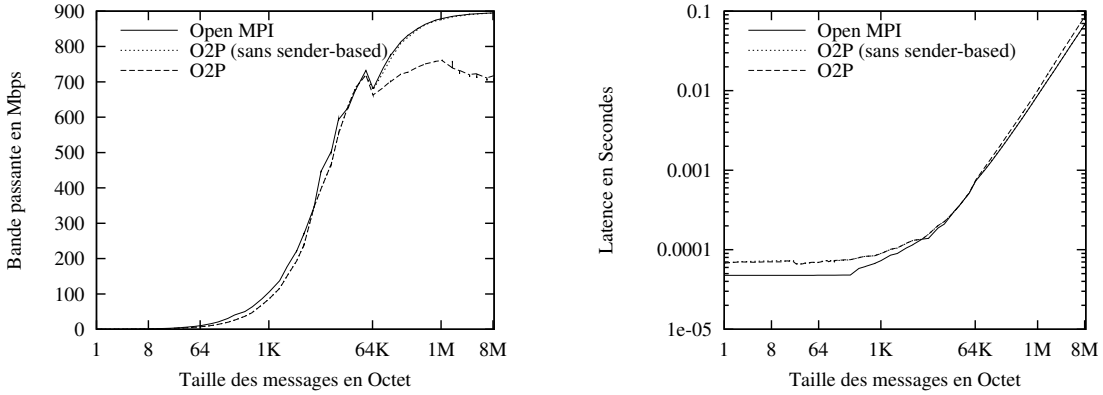



FIG. 5.29 – Bande passante et latence de O2P avec un enregistreur d'événements distribué

Les résultats montrent que les performances de O2P sont toujours très proches des performances originales de Open MPI. Cependant si nous comparons ces résultats avec ceux présentés sur la figure 5.24 obtenus avec un enregistreur d'événements centralisé, nous constatons que les performances sont légèrement moins bonnes notamment en terme de latence pour les petits messages. Cette différence s'explique par le *bruit* créé sur le réseau par les messages que s'échangent les enregistreurs d'événements qui perturbe l'exécution de l'application.

#### 5.3.4.4 Taille des données attachées sur les messages

Pour évaluer la taille des données attachées sur les messages applicatifs avec l'enregistreur d'évènements distribué, nous utilisons 64 machines (26 *paramount* et 38 *paraquad*) et exécutons la classe C des applications du NAS Parallel Benchmark. Les deux paramètres pouvant influencer le comportement de l'enregistreur d'événements distribué sont $f$, le degré de duplication d'un déterminant, et $g$, le degré de diffusion dans l'algorithme épidémique. Les résultats sont des moyennes sur 5 exécutions de chaque application.

Nous commençons par évaluer l'impact du degré de diffusion. Pour cela, nous fixons $f = 1$, c'est à dire que l'application peut tolérer la défaillance d'un nœud, et nous faisons varier le degré de diffusion de 0 à 8. Les résultats de cette expérience sont présentés sur La figure 5.30. Ils montrent que l'enregistreur d'événements distribué parvient à réduire efficacement le nombre d'estampilles attachées sur les messages de l'application et ce avec un degré de diffusion réduit. En effet, dans la plupart des cas, un degré de diffusion de 2 est suffisant pour réduire efficacement le nombre d'estampilles. Nous pouvons voir, de plus, que l'enregistreur d'événements distribué passe à l'échelle puisque nous ne constatons qu'il arrive à réduire le nombre d'estampilles attachées sur les messages même pour les applications composées de 128 processus.

Comme nous utilisons 64 nœuds pour cette expérience, 2 processus sont exécutées sur

chaque nœud durant les expériences avec 128 processus. Nous pouvons alors remarquer l'intérêt d'utiliser un enregistreur d'événements par nœud : un processus est automatiquement au courant des intervalles d'exécution devenant stables pour l'autre processus situé sur le même nœud. Sur la figure 5.30, cela se traduit pour une augmentation moins importante du nombre d'estampilles attachées sur les messages lors du passage de 64 à 128 processus que lors du passage à 32 à 64. Ceci est particulièrement remarquable pour FT et SP, pour lesquelles nous pouvons même constater parfois une baisse du nombre d'estampilles.

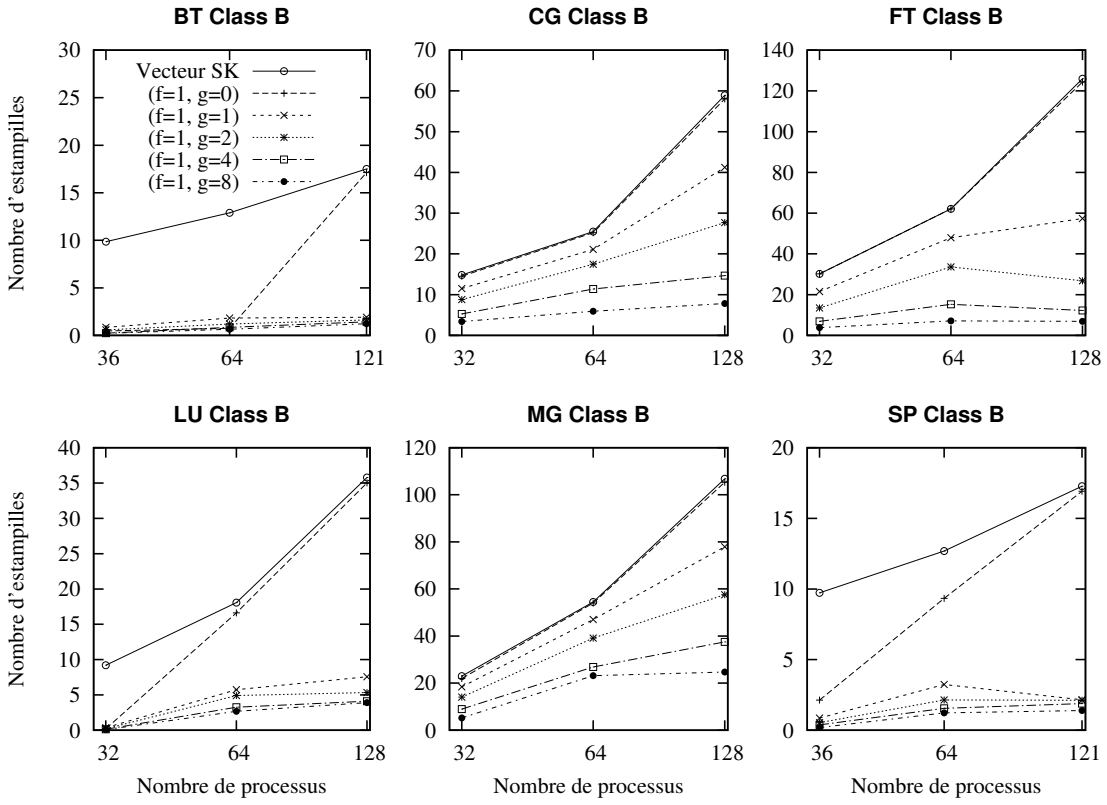



FIG. 5.30 – Influence du degré de diffusion sur la quantité de données attachées sur les messages de l'application avec un enregistreur d'événements distribué

Dans un deuxième temps, nous évaluons l'impact de degré de duplication. Pour cela, nous exécutons à nouveau les applications, en utilisant un degré de diffusion fixé à 2 et en faisant varier le degré de duplication entre 1 et 5. Les résultats, présentés sur la figure 5.31, montrent que le degré de duplication a peu d'influences sur les capacités de O2P à réduire la taille des données à attachées sur les messages. Cependant il a un impact sur les performances des applications. Ainsi pour LU classe B avec 64 processus, le surcoût sur les performances de l'application passe de 20% pour un degré de duplication de 1 à 68% pour un degré de duplication de 5.

#### 5.3.4.5 Performance des applications

Pour évaluer les performances des applications avec l'enregistreur d'événements distribué, nous utilisons 128 machines (38 *paradent*, 29 *paramount*, et 61 *paraquad*) et exécutons les applications du NAS Parallel Benchmark. Les résultats sont des moyennes

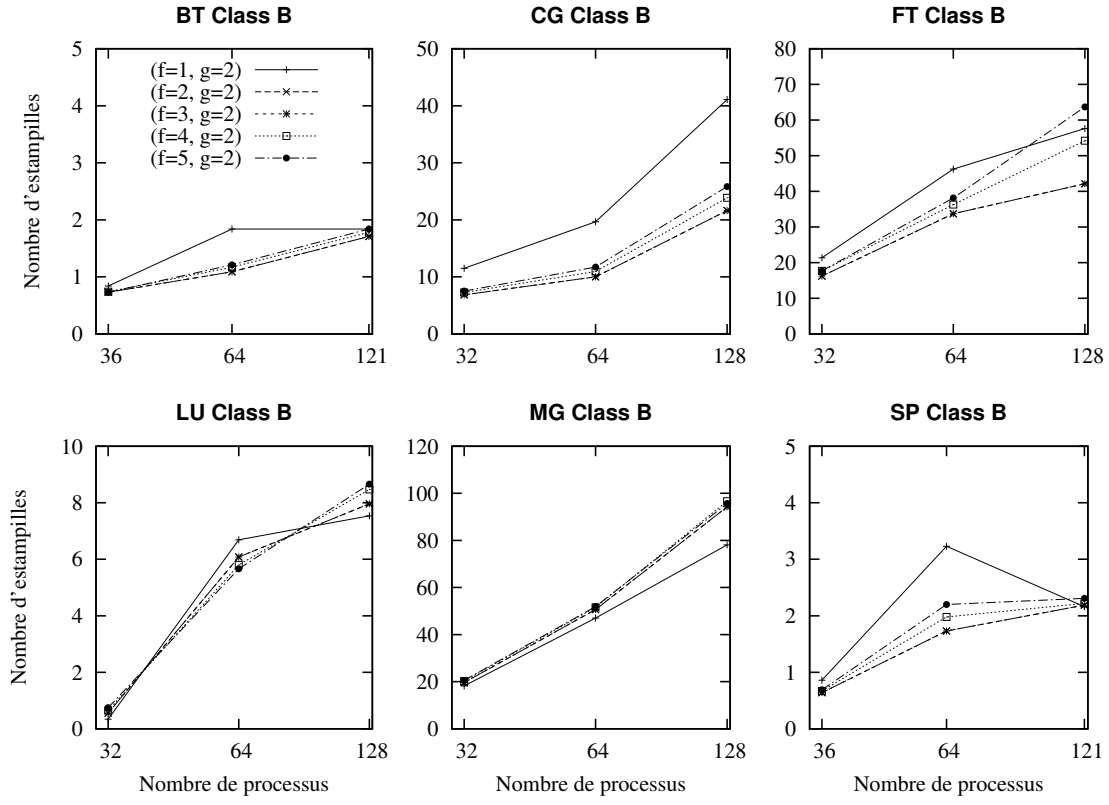



FIG. 5.31 – Influence du degré de duplication sur la quantité de données attachées sur les messages de l'application avec un enregistreur d'événements distribué

sur 8 exécutions de chaque application.

La figure 5.32 compare les performances d'un protocole pessimiste utilisant l'enregistreur d'événements distribué avec un degré du duplication de 1 et le protocole O2P utilisant l'enregistreur d'événements distribué avec le même degré de duplication et un degré de diffusion de 2. Pour le protocole pessimiste, il est inutile de diffuser les informations sur les intervalles d'exécution qui deviennent stable car un message envoyé ne dépend jamais d'aucun intervalle d'exécution non stable. Le degré de diffusion est donc fixé à 0. Nous fixons le degré de duplication à 1 pour une comparaison équitable avec les expériences que nous avons présentées pour l'enregistreur d'événements centralisé. Comme l'enregistreur d'événements centralisé sauvegarde les déterminants en mémoire volatile, on peut considérer qu'il fournit aussi un degré de duplication de 1 pour les déterminants.

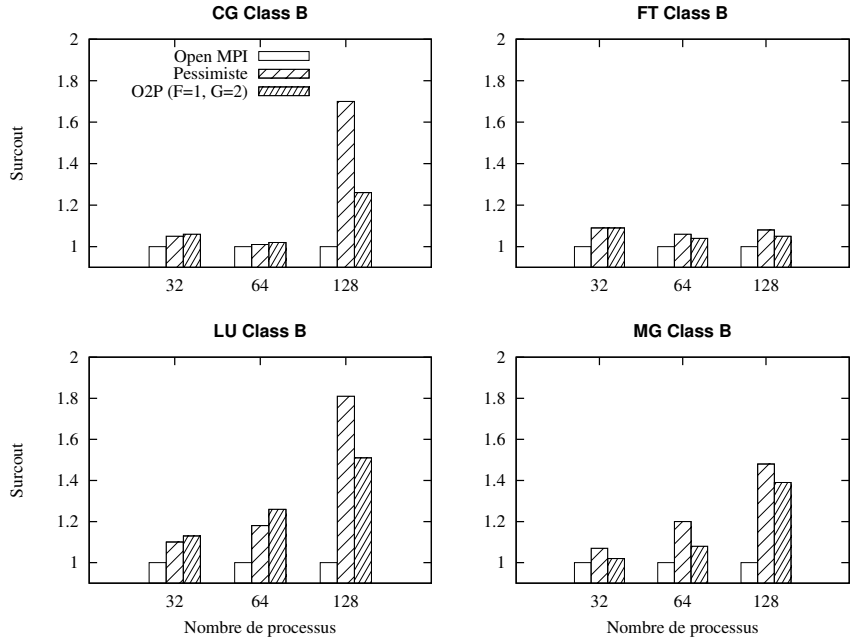



FIG. 5.32 – Performances de O2P et d'un protocole à enregistrement de messages pessimiste avec un enregistreur d'événements distribué

Cette expérience montre tout d'abord, en comparant les résultats présentés sur la figure 5.32 avec les résultats pour un enregistreur d'événements centralisé présentés sur la figure 5.26, que l'enregistreur d'événements distribué assure un meilleur passage à l'échelle des protocoles à enregistrement de messages. Ainsi pour les applications composées de 128 processus, les performances du protocole pessimiste sont meilleures que précédemment. Ceci est particulièrement vrai pour l'application LU, où les communications sont les plus fréquentes : le surcoût avec le protocole pessimiste est maintenant de l'ordre de 80%.

Si les performances de O2P sont toujours meilleures que celles du protocole pessimiste, elles sont moins bonnes qu'avec un enregistreur d'événements centralisé pour CG et MG avec 128 processus. Ceci s'explique par le bruit créé par le protocole épidémique et illustré par la figure 5.33. Lorsque le degré de diffusion de l'algorithme épidémique augmente, les performances de l'application diminuent.

Enfin, la figure 5.34 compare les performances de O2P avec celles d'un protocole optimiste classique auquel serait appliqué l'algorithme de Singhal et Kshemkalyani. Le degré de duplication est toujours de 1. Le degré de diffusion est fixé à 0 pour le protocole op-

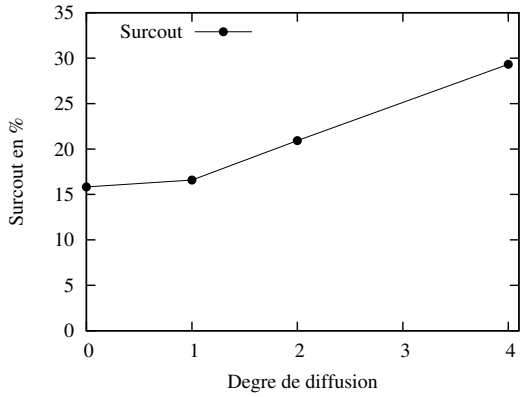



FIG. 5.33 – Influence du degré de diffusion sur les performances de LU classe C avec 128 processus

timiste classique puisqu'il n'exploite pas ces informations et toujours à 2 pour O2P. Les performances des deux protocoles sont à nouveau équivalentes, les applications considérées n'ayant pas une taille suffisante pour que l'enregistrement optimiste actif puisse améliorer les performances des applications.

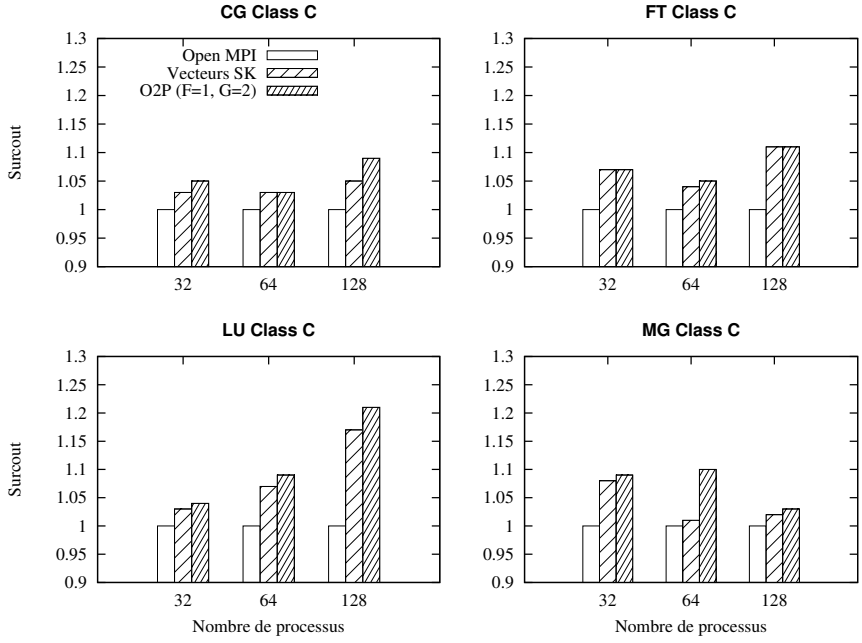



FIG. 5.34 – Performances de O2P et d'un protocole à enregistrement de messages optimiste classique avec un enregistreur d'événements distribué

#### 5.3.4.6 Discussion

La première conclusion que nous pouvons tirer de ces expériences est que notre enregistreur d'événements distribué permet de résoudre le problème de passage à l'échelle dans l'enregistrement des informations de dépendance, que ce soit pour un protocole à

enregistrement de messages pessimiste ou pour O2P.

La comparaison entre O2P et un protocole optimiste *classique* a montré que leurs performances sont équivalentes pour les tailles d'application que nous avons considérées. Nos évaluations ont aussi montré que pour O2P la taille des données attachées sur les messages applicatifs augmentait très peu avec la taille de l'application par rapport à un protocole optimiste classique, en faisant un protocole passant mieux à l'échelle. Il faudrait pouvoir tester les deux protocoles avec des applications de plus grande taille pour voir si l'enregistrement de messages optimiste actif peut réellement offrir de meilleures performances que les protocoles optimistes existants.

Tout en offrant des performances équivalentes à un protocole optimiste *classique*, O2P a quelques avantages supplémentaires. En effet, le nombre inférieur d'estampilles attachées sur les messages applicatifs signifie que chaque processus à plus d'informations sur la stabilité (cf. définition 5.3) des intervalles d'exécution dont il dépend. Ainsi, les performances lors de l'envoi d'un message vers le monde extérieur peuvent être améliorées car le processus émetteur a alors besoin d'obtenir moins d'informations pour savoir quand l'intervalle d'exécution d'émission est valide (cf. définition 5.4). Pour les mêmes raisons, les performances du protocole en recouvrement peuvent être meilleures, les processus non fautifs pouvant déterminer plus vite leur état valide maximum. Ces propriétés seraient à vérifier expérimentalement.

Enfin les expériences ont montré que si l'algorithme épidémique de l'enregistreur d'événements distribué réduit efficacement les quantités de données attachées sur les messages applicatifs, les communications supplémentaires qu'il implique ont un impact sur les performances des applications. Nous voyons deux solutions pour résoudre ce problème. La première solution serait de sélectionner les processus vers lesquels diffuser les informations sur les déterminants stables. En effet, un enregistreur d'événements connaît au moins en partie les dépendances entre les processus de l'application grâce aux déterminants qu'il reçoit. Il pourrait exploiter ces informations pour déterminer à quels processus il est le plus pertinent d'envoyer les informations sur un nouveau déterminant stable. Ainsi, l'algorithme pourrait réduire le nombre total de messages envoyés tout en conservant la même efficacité. La deuxième solution serait de placer les enregistreurs d'événements sur des nœuds sur lesquels ne s'exécutent pas de processus de l'application pour que les communications entre enregistreurs d'événements ne perturbent pas l'exécution de l'application. Cette solution a deux inconvénients. Le premier est que cette solution implique l'utilisation de nœuds supplémentaires pour assurer la tolérance aux fautes des applications. Le deuxième est que cette solution ne permet plus le partage de la connaissance des états stables entre processus s'exécutant sur le même nœud.

## 5.4 Synthèse

O2P est un protocole à enregistrement de messages optimiste *actif* visant à résoudre les problèmes de passage à l'échelle des protocoles de recouvrement arrière. L'enregistrement de messages optimiste actif consiste à sauvegarder les informations de dépendances entre les processus de l'application sur support stable au plus tôt pour limiter la taille des données à attacher sur les messages applicatifs. Cette stratégie permet de plus de limiter les risques de création de processus orphelins et améliore les performances lorsque des messages sont à envoyer vers le monde extérieur. Nous avons prouvé que O2P est capable de tolérer plusieurs fautes simultanées de processus.

Nous avons mis en œuvre O2P dans la bibliothèque Open MPI et l'avons évalué sur la plate-forme Grid'5000. Nos expériences ont montré que l'enregistreur d'événements cen-

tralisé généralement considéré dans les travaux récents sur les protocoles à enregistrement de messages ne passait pas à l'échelle. C'est pourquoi nous avons proposé un nouveau modèle d'enregistreur d'événements distribué.

Cet enregistreur d'événements, que nous avons mis en œuvre dans Open MPI, exploite la mémoire des nœuds sur lesquels s'exécute l'application pour enregistrer les déterminants. Il est fondé sur un algorithme épidémique pour diffuser les informations sur les nouveaux déterminants stables aux processus de l'application.

Les évaluations de O2P et d'un protocole à enregistrement de messages pessimiste avec l'enregistreur d'événements distribué montrent qu'il permet à ces protocoles de mieux passer à l'échelle. Les résultats de nos expériences montrent d'ailleurs que la combinaison de O2P avec l'enregistreur d'événements distribué fournissent une solution de recouvrement arrière passant à l'échelle. Ces résultats seraient cependant à confirmer avec des applications de plus grande taille que celles que nous avons utilisées pour nos expériences.

Enfin nos expériences ont montré l'efficacité de l'algorithme épidémique pour réduire la taille des données attachées par O2P sur les messages applicatifs, mais elles ont aussi mis en évidence l'impact non négligeable de celui-ci sur les performances des applications. Des solutions devraient être étudiées pour limiter ce surcoût tout en conservant l'efficacité de l'algorithme. Pour cela, les enregistreurs d'événements pourraient exploiter les informations sur les dépendances entre les processus de l'application, qu'ils connaissent au travers des déterminants qu'ils reçoivent, pour mieux sélectionner les processus vers lesquels ils diffusent leurs informations sur les nouveaux déterminants stables.

# Conclusion

Une grille de calcul est un système distribué qui regroupe un très grand nombre de ressources de calcul hétérogènes, pouvant appartenir à différents domaines d'administration et distribuées géographiquement. Les grilles de calcul sont attractives car elles peuvent fournir à leurs utilisateurs les ressources nécessaires à l'exécution d'applications de calcul scientifique. En effet, plus la quantité de ressources disponible pour ce type d'application est grande, plus les résultats fournis peuvent être précis, permettant aux scientifiques de mieux comprendre des phénomènes complexes.

Les ressources d'une grille de calcul sont volatiles. Cette volatilité est due aux ajouts et retraits volontaires de ressources par leurs propriétaires et aux défaillances. Étant donné le grand nombre de ressources pouvant composer une grille, les risques de défaillances matérielles sont élevés. La durée d'une application de calcul scientifique pouvant être de plusieurs jours, voire de plusieurs mois, la probabilité qu'elle subisse une défaillance compromettant sa bonne terminaison est très élevée.

Les grilles de calcul ne peuvent être une solution attractive pour l'exécution d'applications de calcul scientifique que si les utilisateurs parviennent à obtenir les résultats pour les applications qu'ils soumettent, dans un délai acceptable, et avec des efforts raisonnables. C'est pourquoi des solutions doivent être trouvées pour assurer l'exécution fiable d'applications distribuées dans les grilles de calcul. Ces solutions passent par des mécanismes de tolérance aux fautes et d'auto-réparation.

La tolérance aux fautes, et en particulier la tolérance aux fautes pour les applications distribuées, est un domaine très actif de la recherche en informatique. Dans ce document nous nous sommes concentrés sur la recherche de solutions de tolérance aux fautes adaptées aux grilles de calcul, c'est-à-dire adaptées à leur taille, leur volatilité et leur hétérogénéité. Nous avons proposé un ensemble de solutions pour assurer l'exécution fiable d'applications distribuées sur grille de calcul, visant de plus à faire des grilles de calcul un environnement d'exécution simple à utiliser et performant.

## Contributions

Pour répondre à ces objectifs, nos travaux se sont articulés autour de trois axes de recherche :

- La conception d'un service pour le redémarrage automatique d'applications défaillantes à l'aide de techniques de recouvrement arrière,
- la conception, la mise en œuvre et l'évaluation d'un cadre pour la haute disponibilité et l'auto-réparation de services de grille,
- la conception, la mise en œuvre et l'évaluation de protocoles de recouvrement arrière passant à l'échelle.

Ces travaux fournissent un ensemble de contributions permettant d'assurer l'exécution fiable d'applications distribuées dans la grille.

### XtreemGCP : un service de recouvrement arrière pour applications distribuées

XtreemGCP [127] est un service de grille que nous avons conçu dans le cadre du projet européen XtreemOS. Il a été mis en œuvre par les partenaires XtreemOS de Düsseldorf et de l'INRIA.

XtreemGCP applique des techniques de recouvrement arrière pour redémarrer automatiquement les applications subissant des défaillances. Pour cela, il prend en charge les interactions nécessaires avec les autres services de XtreemOS, et notamment avec le service d'allocation de ressources pour obtenir les ressources nécessaires pour remplacer les ressources défaillantes. Il tire aussi profit de XtreemFS pour stocker les données de sauvegarde de points de reprise.

Fondé sur une architecture complètement distribuée passant à l'échelle, XtreemGCP est capable de fournir des solutions de recouvrement arrière de manière transparente pour les applications distribuées. Ainsi les utilisateurs n'ont pas à modifier leurs applications pour pouvoir profiter de ce service.

XtreemGCP est un service générique. Il permet d'intégrer différentes solutions de recouvrement arrière pour mieux répondre aux besoins des utilisateurs. Nous avons proposé une interface générique pour les outils de sauvegarde de points de reprise de processus permettant à XtreemGCP d'exploiter plusieurs de ces outils de manière simple [99, 143]. Ainsi XtreemGCP est capable de prendre en charge des applications exécutées sur des ressources hétérogènes. De la même manière, XtreemGCP offre les briques de base pour la mise en œuvre de différents protocoles de recouvrement arrière [127, 68]. De plus, il peut exploiter des mécanismes de recouvrement arrière fournis par des bibliothèques tolérantes aux fautes telles que Open MPI. Tout ceci assure que dans le futur XtreemGCP puisse exploiter les nouveaux résultats de la recherche en tolérance aux fautes.

### Semias : un cadre pour la mise en œuvre de services de grille hautement disponibles et auto-réparants

Semias permet de rendre des services de grille, tels que XtreemGCP, hautement disponibles et auto-réparants. L'intégration d'un service à Semias ne nécessite que peu d'efforts de la part du programmeur car Semias est fondé sur la duplication active des services. La mise en œuvre de la duplication active au dessus d'un réseau logique structuré rend les défaillances et les reconfigurations de services dupliqués complètement transparentes pour les utilisateurs.

Le système de communication de groupes de Semias, fondé sur l'architecture proposée par Mena et al. [128] pour les groupes dynamiques, permet de dissocier la suspicion d'un nœud de son éviction du groupe. Ceci fait de Semias un système bien adapté pour les réseaux étendus où les latences peuvent être importantes et variables car son fonctionnement n'est pas altéré par les fausses suspicions.

Semias fournit à notre connaissance la première mise en œuvre de duplication active dans un réseau logique structuré. Pour gérer de manière efficace la volatilité des nœuds, Semias exploite un module de supervision chargé de rassembler des informations sur l'état du réseau logique et des groupes de duplicatas. Ce module de supervision tente ensuite de prendre les meilleures décisions d'auto-réparation possibles pour assurer la disponibilité des services tout en minimisant le nombre total de reconfigurations.

Un prototype de Semias a été mis en œuvre et utilisé pour rendre le service de gestion d'applications du système de grille Vigne hautement disponible. Les évaluations, menées sur la plate-forme Grid'5000, ont montré que Semias offrait de performances acceptables

aux services dupliqués. De plus, elles ont montré que les mécanismes d'auto-réparation de Semias assuraient la haute disponibilité des services dans un environnement dynamique, tout en limitant le nombre de reconfigurations.

### O2P : un protocole à enregistrement de messages optimiste actif pour applications à échange de messages de grande taille

O2P [162] est un protocole de recouvrement arrière pour applications à échange de message garantissant de bonnes performances même pour des applications composées d'un grand nombre de processus.

O2P est fondé sur l'enregistrement de messages optimiste *actif*. Ce nouveau protocole que nous proposons, vise à minimiser les risques de création de processus orphelins et à optimiser les performances en fonctionnement normal des applications en enregistrant les informations de dépendance entre les processus de l'application sur support stable au plus tôt. Ainsi cette stratégie permet à O2P de minimiser les quantités d'information de dépendance à attacher sur les messages applicatifs par rapport aux protocoles optimistes existants. Dans ce document, nous prouvons que ce protocole permet de tolérer plusieurs fautes simultanées de processus.

Pour assurer un meilleur passage à l'échelle des protocoles à enregistrement de messages, nous avons de plus proposé une solution pour la gestion distribuée de la sauvegarde des informations de dépendance. Cette solution exploite la mémoire vive des nœuds sur lesquels sont exécutés les processus de l'application et se fonde sur un algorithme épidémique pour diffuser les informations sur les dépendances sauvegardées.

O2P a été mis en œuvre dans la bibliothèque Open MPI et évalué sur Grid'5000. Nous avons montré que O2P était capable de réduire efficacement la taille des données attachées sur les messages de l'application. De plus, nous avons montré que notre gestion distribuée des informations de dépendance passait mieux à l'échelle que la solution centralisée généralement considérée. La combinaison de ces deux contributions offre un protocole de recouvrement arrière passant à l'échelle pour les applications de grande taille fondées sur le paradigme de communication par échange de messages.

Cette thèse présente donc un ensemble de contributions adaptées aux caractéristiques des grilles de calcul, pour assurer l'exécution fiable des applications distribuées de calcul scientifique. XtreemGCP assure à l'utilisateur l'obtention de ses résultats en appliquant de manière automatique et transparente des techniques de recouvrement arrière. XtreemGCP peut utiliser O2P pour appliquer aux applications à échange de messages de grande taille un protocole de recouvrement arrière passant à l'échelle. Enfin, Semias peut être utilisé pour assurer le bon fonctionnement de XtreemGCP en lui apportant les propriétés de haute disponibilité et d'auto-réparation.

## Perspectives

Les travaux présentés dans ce document ouvrent de nombreuses perspectives de recherche. Nous commençons par présenter les perspectives à court terme avant d'étendre sur des perspectives à plus long terme.

### Évaluation du protocole O2P au redémarrage

Les limites actuelles de Open MPI ne nous ont pas permis d'évaluer les performances de O2P lors du redémarrage après une ou plusieurs défaillances. Les caractéristiques du

protocole O2P font que le risque de création de processus orphelins est très faible. Les performances de O2P au redémarrage devraient donc être bonnes. Cependant il serait important d'évaluer l'impact de la gestion distribuée de l'enregistrement des messages sur celles-ci.

### Un protocole à enregistrement de messages hiérarchique

Dans le cas d'une grille de calcul de type fédération de grappes de calcul, il est envisageable qu'une application à échange de messages soit distribuée sur plusieurs grappes. Dans ce cas, le contexte d'exécution de l'application est hétérogène notamment au niveau de la latence : la latence entre deux nœuds d'une même grappe de calcul est beaucoup plus faible que celle entre deux nœuds de grappes distantes. Des travaux tentent d'optimiser les performances des applications pour ce cas d'exécution en limitant au maximum les communications entre processus de grappes distantes [125].

Dans O2P, lors du redémarrage après une défaillance, les processus fautifs doivent communiquer avec tous les processus de l'application. Dans le cas d'une application distribuée sur une fédération de grappes de calcul, les performances au redémarrage seraient limitées par la latence élevée entre certains processus de l'application. C'est pourquoi nous pensons que pour adapter O2P à ce contexte, il pourrait être combiné avec un protocole à enregistrement de messages pessimiste dans un protocole hiérarchique [161]. O2P serait utilisé pour les communications au sein d'une grappe de calcul pour offrir les meilleurs performances possibles en fonctionnement sans défaillance. Le protocole pessimiste serait utilisé pour les communications entre grappes de calcul pour assurer l'indépendance entre les processus exécutés sur différentes grappes, et ainsi éviter les interactions entre processus distants au redémarrage.

### Étendre l'utilisation de Semias

Nous avons utilisé Semias pour rendre le service de gestion d'applications de Vigne hautement disponible et auto-réparant. Son utilisation pourrait être étendue à d'autres services de Vigne, tel que le service de supervision des applications [160]. De même, il serait intéressant d'étudier comment les propriétés de Semias pourraient être exploitées dans le contexte de XtreemOS.

### Exploitation des architectures multi-cœurs

Les processeurs actuels sont fondés sur des architectures multi-cœurs. Dans le futur le nombre de cœurs par processeur risque encore d'augmenter. Certains de ces cœurs pourraient être utilisés pour exécuter un protocole de recouvrement arrière en parallèle avec l'exécution des processus de l'application pour en limiter l'impact sur les performances de l'application. C'est ce que nous avons tenté d'exploiter en proposant l'enregistreur d'événement distribué pour O2P. Dans notre prototype, la gestion de la synchronisation sur un nœud entre les processus de l'application et le processus exécutant le protocole de recouvrement arrière est très simple et ne prend pas en compte par exemple les problèmes de gestion des caches. Une exploitation efficace de ces architectures pourrait permettre de mettre en œuvre des solutions de tolérance aux fautes ayant un impact faible sur les performances des applications en fonctionnement normal.

### Adaptation des mécanismes de tolérance aux fautes

Nos travaux montrent que de nombreux paramètres peuvent influencer le comportement des mécanismes de tolérance aux fautes, comme la fréquence des défaillances ou les caractéristiques des applications. Il faudrait être capable d'adapter la stratégie de tolérance aux fautes au contexte d'exécution pour une plus grande efficacité.

XtreemGCP offre la possibilité de mettre en œuvre différents protocoles de recouvrement arrière fondés sur la sauvegarde de points de reprise coordonnés ou non coordonnés, avec ou sans enregistrement de messages. XtreemGCP pourrait superviser l'exécution des applications pour choisir le protocole de recouvrement arrière le mieux adapté. Pour cela, il pourrait aussi prendre en compte la fréquence des défaillances dans le système. Pour atteindre cet objectif, il faudrait être capable de changer de protocole de recouvrement arrière au cours de l'exécution de l'application. C'est un problème qui n'a encore jamais été étudié à notre connaissance.

Au sein même du protocole O2P, nous avons vu que les paramètres définissant le degré de duplication et le comportement de l'algorithme épidémique de l'enregistreur d'événements distribué influencent les performances du protocole. XtreemGCP pourrait être utilisé pour choisir en fonction du contexte les meilleurs paramètres pour O2P.

Enfin, dans la version actuelle de Semias, le degré de duplication des services est fixe. Le module de supervision de Semias pourrait évaluer la volatilité des nœuds dans le système pour adapter le degré de duplication et les règles d'auto-réparation des services en conséquence, et ainsi assurer la haute disponibilité des services dupliqués même en cas de fréquence des défaillances élevée.

### Un service d'information sur les défaillances

Dans un système distribué tel qu'une grille de calcul, tous les outils utilisés pour la tolérance aux fautes ont besoin d'informations sur l'état du système, et en particulier sur les défaillances. Par exemple, nous avons vu précédemment que XtreemGCP et Semias pourraient exploiter ces informations pour prendre des décisions d'adaptation.

De plus, différents mécanismes de tolérance aux fautes sont en général disponibles dans ces systèmes distribués. Nous avons vu dans Semias que les algorithmes de maintenance des tables routage d'un réseau logique structuré peuvent être utilisés comme détecteur de défaillances. De même les bibliothèques MPI ont en général leurs propres mécanismes pour détecter les défaillances.

C'est pourquoi nous pensons qu'un service chargé de fournir des informations sur les défaillances aux autres services du système pourrait être une brique de base dans la conception de systèmes tolérants aux fautes. Ce service serait chargé de regrouper les informations provenant des différents détecteurs de défaillances présents dans le système pour pouvoir les fournir aux services en ayant besoin. En analysant les informations provenant de différents détecteurs de défaillance, il pourrait fournir des informations plus précises non seulement sur la présence de défaillances mais aussi sur leur fréquence. Les services de tolérance aux fautes du système, ou tout autre service intéressé par les informations sur les défaillances, pourraient alors s'abonner au service et définir les nœuds pour lesquels ils voudraient obtenir des informations et avec quelle précision. Le principal défi associé à la mise en œuvre d'un tel service serait le passage à l'échelle. Nous avons commencé à étudier cette problématique dans le cadre d'une collaboration avec Catalin Leordeanu, doctorant à Politehnica University of Bucarest (Roumanie).

# Bibliographie


[1] Transmission control protocol. RFC 793.

[2] David Abramson, Amanda Lynch, Hiroshi Takemiya, Yusuke Tanimura, Susumu Date, Haruki Nakamura, Karpjoo Jeong, Suntae Hwang, Ji Zhu, Zhong-hua Lu, Celine Amoreira, Kim Baldridge, Hurng-Chun Lee, Chi-Wei Wang, Horng-Liang Shih, Tomas Molina, Wilfred W. Li, and Peter W. Arzberger. Deploying Scientific Applications to the PRAGMA Grid Testbed : Strategies and Lessons. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, pages 241–248, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Adnan Agbaria and Roy Friedman. Starfish : Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)*, page 31, Washington, DC, USA, 1999. IEEE Computer Society.

[4] Adnan Agbaria and James S. Plank. Design, implementation, and performance of checkpointing in netsolve. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN '00)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[5] The Globus Alliance. www.globus.org.

[6] Lorenzo Alvisi and Keith Marzullo. Message Logging : Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2) :149–159, 1998.

[7] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Asanka de Mel, and Elmootazbellah Elnozahy. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, pages 242–249, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Abdelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder Kaur Chouhan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, and Alan Su. DIET : New Developments and Recent Results. In *Euro-Par 2006 Workshops - Parallel Processing*, pages 150–170, 2006.

[9] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks : Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2) :18–28, Feb 1996.

[10] David P. Anderson. BOINC : A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[11] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home : An Experiment in Public-Resource Computing. *Communication of the ACM*, 45(11) :56–61, 2002.

[12] Ali Anjomshoaa, Fred Brisard, Michel Dresher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher, and Andreas Savva. Job Submission Description Language (JSDL) Specification, V. 1.0. Technical Report GFD-R.136, Open Grid Forum, 2008.

[13] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP : Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, 2009.

[14] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation : Practice and Experience*, 18(13) :1705–1723, November 2006.

[15] R.T. Aulwes, D.J. Daniel, N.N. Desai, R.L. Graham, L.D. Risinger, M.A. Taylor, T.S. Woodall, and M.W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 15–25, 2004.

[16] D. Bailey, T. Harris, W. Saphir, R. van der Wilngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report Report NAS-95-020, NASA Ames Research Center, 1995.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[18] Udo Bartlang and Jorg P. Muller. DhtFlex : A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays. In *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pages 377–384, 2008.

[19] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, pages 635–644, June 2003.

[20] Bharat Bhargava and Shu-Renn Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-An Optimistic Approach. In *7th Symposium on Reliable Distributed Systems*, pages 3–12, Oct 1988.

[21] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12) :37–53, 1993.

[22] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *Communications of the ACM*, 5(1) :47–76, 1987.

[23] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1) :47–67, 2003.

[24] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[25] Anita Borg, Jim Baumbach, and Sam Glazer. A Message System Supporting Fault Tolerance. *SIGOPS Operating Systems Review*, 17(5) :90–99, 1983.

[26] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[27] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the Message Logging Model for High Performance. In *International Supercomputer Conference (ISC 2008)*, Dresden, Germany, June 2008.

[28] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In *Proceedings of the 19th IEEE InternationalParallel and Distributed Processing Symposium (IPDPS'05)*, volume 1, page 97, Los Alamitos, CA, USA, April 2005. IEEE Computer Society.

[29] Aurelien Bouteiller and Frederic Desprez. Fault Tolerance Management for a Hierarchical GridRPC Middleware. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)*, pages 484–491, Washington, DC, USA, 2008. IEEE Computer Society.

[30] Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V Project : A Multiprotocol Automatic Fault-Tolerant MPI . *International Journal of High Performance Computing Applications*, 20(3) :319–333, 2006.

[31] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated Checkpoint Versus Message Log for Fault Tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2003)*, pages 242–250, 2003.

[32] Aurelien Bouteiller, Thomas Ropars, George Bosilca, Christine Morin, and Jack Dongarra. Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[33] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2 : a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03)*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated Application-Level Checkpointing of MPI Programs. *ACM SIGPLAN Notices*, 38(10) :84–94, 2003.

[35] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and Dynamic Software Composition with Sharing. In *International Workshop on Component-Oriented Programming (WCOP '02)*, Malaga, Spain, June 2002.

[36] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems (2nd Edition)*, pages 199–216, 1993.

[37] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Marti n, Gregory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler With High Level Components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[38] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems : Xtrem Web architecture, programming models, security, tests and convergence with grid. *Future Generation Computer Systems*, 21(3) :417–437, 2005.

[39] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. A Peer-to-Peer Extension of Network-Enabled Server Systems. In *1st International Conference on e-Science and Grid Computing (e-Science 2005)*, pages 430–437, Melbourne, Australia, December, 5-8 2005. IEEE.

[40] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4) :685–722, 1996.

[41] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225–267, 1996.

[42] K.Mani Chandy and Leslie Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1) :63–75, 1985.

[43] Marc Chérèque, David Powell, Phillipe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. In *Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 28–37, 1992.

[44] Renaud Lottiaux Christine Morin, Pascal Gallard and Geoffroy Vallée. Towards an Efficient Single System Image Cluster Operating System. *Future Generation Computer Systems*, 20(2), January 2004.

[45] The Large Hadron Collider. http ://public.web.cern.ch/public/en/lhc/lhc-en.html.

[46] The XtreemOS Consortium. Design and Implementation of Basic Checkpoint/Restart Mechanisms in Linux. Technical Report D2.1.3, XtreemOS, 2007.

[47] The XtreemOS Consortium. Design of the Architecture for Application Execution Management. Technical Report D3.3.2, XtreemOS, 2007.

[48] World Wide Web Consortium. Web services description language (wsdl) 1.1. http ://www.w3.org/TR/wsdl.html.

[49] Massimo Coppola, Yvon Jégou, Brian Matthews, Christine Morin, Luis Pablo Prieto, Óscar David Sánchez, Erica Y. Yang, and Haiyan Yu. Virtual Organization Support within a Grid-Wide Operating System. *IEEE Internet Computing*, 12(2) :20–28, 2008.

[50] Toni Cortes, Carsten Franke, Yvon Jégou, Thilo Kielmann, Domenico Laforenza, Brian Matthews, Christine Morin, Luis Pablo Prieto, and Alexander Reinefeld. XtreemOS : a Vision for a Grid Operating System. Technical report, XtreemOS, 2008.

[51] Christopher Dabrowski. Reliability in grid computing systems. *Concurrency and Computation : Practice and Experience*, 21 :927–959, 2009.

[52] Leonardo Dagum and Ramesh Menon. OpenMP : An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1) :46–55, Jan-Mar 1998.

[53] Om P. Damani and Vijay K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. In *International Conference on Distributed Computing systems*, pages 108–115, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[54] Om P. Damani, Yi-Min Wang, and Vijay K. Garg. Distributed Recovery with K-optimistic Logging. *Journal of Parallel and Distributed Computing*, 63 :1193–1218, 2003.

[55] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. In *Proceedings of the 2nd workshop on Middleware for grid computing (MGC'04)*, pages 35–40, New York, NY, USA, 2004. ACM.

[56] Xavier Défago, André Schiper, and Nicole Sergent. Semi-Passive Replication. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[57] Xavier Defago, Andre Schiper, and Peter Urban. Total order broadcast and multicast algorithms : Taxonomy and survey. *ACM Computing Surveys*, 36(4) :372–421, 2004.

[58] DEISA. http ://www.deisa.eu.

[59] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing(PODC '87)*, pages 1–12, New York, NY, USA, 1987. ACM.

[60] Samir Djilali, Thomas Herault, Oleg Lodygensky, Tangui Morlier, Gilles Fedak, and Franck Cappello. RPC-V : Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC2004)*, pages 39–39, November 2004.

[61] Jörg Domaschka, Franz J. Hauck, and Hans P. Reiser. Deterministic Multithreading for Java-Based Replicated Objects. In *18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*, pages 13–15, Dallas, Texas, USA, 2006.

[62] Niels Drost, Rob V. van Nieuwpoort, and Henri Bal. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, page 14, Washington, DC, USA, 2006. IEEE Computer Society.

[63] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3) :375–408, 2002.

[64] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho : Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5) :526–531, 1992.

[65] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On The Use And Implementation Of Message Logging. In *24th International Symposium On Fault-Tolerant Computing (FTCS-24)*, pages 298–307. IEEE Computer Society, 1994.

[66] Sun Grid Engine. http ://www.sun.com/software/sge.

[67] Graham E. Fagg and Jack Dongarra. FT-MPI : Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[68] Eugen Feller. Independent Checkpointing in a Heterogeneous Grid Environment. Master's thesis, Heinrich Heine University of Duesseldorf, Germany, November 2009.

[69] Martin Feller, Ian Foster, and Stuart Martin. GT4 GRAM : A Functionality and Performance Study. Technical report, www.globus.org, 2007.

[70] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM*, 32(2) :374–382, 1985.

[71] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard. www.mpi-forum.org.

[72] The Open Grid Forum. http ://www.ogf.org.

[73] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. Technical Report GFD-R.80, Open Grid Forum, 2006.

[74] Ian Foster. Globus Toolkit Version 4 : Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4), 2006.

[75] Ian Foster and Carl Kesselman. Computational grids. *The grid : blueprint for a new computing infrastructure*, pages 15–51, 1999.

[76] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid : Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3) :200–222, 2001.

[77] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-G : A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3) :237–246, 2002.

[78] H. Kohmann G. Schneider and H. Bugge. Fault Tolerant Checkpointing Solution for Clusters and Grid Systems. Technical report, HPC4U project, 2007.

[79] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[80] Yolanda Gil, Pedro A. González-Calero, and Ewa Deelman. On the Black Art of Designing Computational Workflows. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science (WORKS '07)*, pages 53–62, New York, NY, USA, 2007. ACM.

[81] Cedric Le Goater, Daniel Lezcano, Clement Calmels, Dave Hansen, Serge Hallyn, and Hubertus Franke. Making applications mobile using containers. In *Proceedings of the Ottawa Linux Symposium*, pages 347–367, 2006.

[82] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. SAGA : A Simple API for Grid Applications – High-Level ApplicationProgramming on the Grid. *Computational Methods in Science and Technology*, 12(1), May 2006.

[83] LHC Computing Grid. http ://lcg.web.cern.ch/lcg/default.htm.

[84] Grid'5000. http ://www.grid5000.fr.

[85] Vassos Hadzilacos and Sam Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed systems (2nd Edition)*, pages 97–145, 1993.

[86] Mark Hayden. The ensemble system. Technical Report TR98-1662, Cornell University, Ithaca, NY, USA, 1998.

[87] Jean-Michel Hélary, Achour Mostefaoui, Robert H.B. Netzer, and Michel Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, 13(1) :29–43, 2000.

[88] Erik Hendriks. Bproc : The beowulf distributed process space. In *Proceedings of the 16th international conference on Supercomputing (ICS '02)*, pages 129–136, New York, NY, USA, 2002. ACM.

[89] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen Shipman, and Andrew Lumsdaine. A Case for Standard Non-Blocking Collective Operations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[90] HPC4U. www.hpc4u.org.

[91] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. XtreemFS - a case for object-based file systems in Grids. *Concurrency and Computation : Practice and Experience*, 20(8), 2008.

[92] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A Checkpoint and Restart Service Specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

[93] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

[94] Soonwook Hwang and Carl Kesselman. A Flexible Framework for Fault Tolerance in the Grid . *Journal of Grid Computing*, 1(3) :251–272, 2003.

[95] Infiniband. http ://www.infinibandta.org.

[96] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, and Jozsef Kovacs. *Grid Middleware and Services - Challenges and Solutions*, chapter Improvement on Fault-Tolerance and Migration Facilities within the Grid Computing Environment - Integration With The Low-Level Checkpointing Packages, pages 305–317. CoreGRID, springer us edition, 2008.

[97] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, Maciej Stroinski, Jozsef Kovacs, and Attila Kertesz. Grid Checkpointing Architecture - Integration of low-level checkpointing capabilites with GRID. Technical Report TR-0075, CoreGRID, 2007.

[98] Gracjan Jankowski, Jozsef Kovacs, Norbert Meyer, Radoslaw Januszewski, and Rafal Mikolajczak. Towards Checkpointing Grid Architecture. *Parallel Processing and Applied Mathematics*, 3911 :659–666, 2006.

[99] Paul Hargrove Jason Duell and Eric Roman. The design and implementation of Berkeley Lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2005.

[100] Emmanuel Jeanvoine. *Intergiciel pour l exécution efficace et fiable d applications distribuées dans des grilles dynamiques de très grande taille*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, November 2007. In french.

[101] Emmanuel Jeanvoine and Christine Morin. RW-OGS : an Optimized Random walk Protocol for Resource Discovery in Large Scale Dynamic Grids. In *9th IEEE/ACM International Conference on Grid Computing*, pages 168–175, Oct 2008.

[102] Emmanuel Jeanvoine, Christine Morin, and Daniel Leprince. Vigne : Executing Easily and Efficiently a Wide Range of Distributed Applications in Grids. In *Proceedings of Euro-Par 2007*, pages 394–403, Rennes, France, 2007.

[103] Emmanuel Jeanvoine, Louis Rilling, Christine Morin, and Daniel Leprince. Using Overlay Networks to Build Operating System Services for Large Scale Grids. In *Proceedings of the 5th International Symposium on Parallel and Distributed Computing (ISPDC 2006)*, pages 191–198, Timisoara, Romania, July 2006.

[104] Yvon Jégou. Dynamic Memory Management on Mome DSM. In *Proc. Intl. Workshop on Distributed Shared Memory on Clusters (DSM 2006)*, Singapore, May 2006. Held in conjunction with CCGrid 2006.

[105] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Digest of Papers : The 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[106] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11(3) :462–491, 1990.

[107] Bob Jones. An Overview of the EGEE Project. In *Peer-to-Peer, Grid, and Service-Orientation in Digital Library Architectures*, volume 3664 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 2005.

[108] Oren Laadan and Jason Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[109] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7) :558–565, 1978.

[110] Leslie Lamport. The Part-Time Parliament. *ACM Transactions in Computer System*, 16(2) :133–169, 1998.

[111] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2) :79–103, 2006.

[112] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3) :382–401, 1982.

[113] Butler W. Lampson and Howard E. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Xerox Palo Alto Research Center, 1979.

[114] Jean-Claude Laprie. Dependable Computing and Fault Tolerance : Concepts and Terminology. In *Proceedings of the 15th IEEE Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, June 1985.

[115] Byoungjoo Lee, Taesoon Park, Heon Y. Yeom, and Yookun Cho. An Efficient Algorithm for Causal Message Logging. In *IEEE Symposium on Reliable Distributed Systems*, pages 19 – 25, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[116] Avraham Leff, James T. Rayfield, and Daniel M. Dias. Service-Level Agreements and Commercial Grids. *IEEE Internet Computing*, 7(4) :44–50, 2003.

[117] Pierre Lemarinier, Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, and Franck Cappello. Improved Message logging versus Improved Coordinated Checkpointing for Fault Tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004.

[118] Michael J. Lewis, Adam J. Ferrari, Marty A. Humphrey, John F. Karpovich, Mark M. Morgan, Anand Natrajan, Anh Nguyen-Tuong, Glenn S. Wasson, and Andrew S. Grimshaw. Support for Extensibility and Site Autonomy in the Legion Grid System Object Model. *Journal of Parallel and Distributed Computing*, 63(5) :525–538, 2003.

[119] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[120] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor-A Hunter of Idle Workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[121] Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evrepidou. MPI-FT : Portable Fault Tolerance Scheme for MPI. *Parallel Processing Letters*, 10(4) :371–382, 2000.

[122] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (Supercomputing '95)*, page 37, New York, NY, USA, 1995. ACM.

[123] André Luckow and Bettina Schnor. Migol : A fault-tolerant service framework for MPI applications in the grid. *Future Generation Computer Systems*, 24(2) :142 – 152, 2008.

[124] André Luckow and Bettina Schnor. Service Replication in Grids : Ensuring Consistency in a Dynamic, Failure-Prone Environment. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–7, April 2008.

[125] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. Efficient MPI Collective Operations for Clusters in Long-and-Fast Networks. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[126] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[127] John Mehnert-Spahn, Thomas Ropars, Michael Schoettner, and Christine Morin. The Architecture of the XtreemOS Grid Checkpointing Service. In *15th International Euro-Par Conference*, pages 429–441, Delft, The Netherlands, August 2009.

[128] Sergio Mena, André Schiper, and Pawed Wojciechowski. A Step towards a New Generation of Group Communication Systems. In *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432, Rio de Janeiro, Brazil, 2003.

[129] Christine Morin. XtreemOS : A Grid Operating System Making your Computer Ready for Participating in Virtual Organizations. In *ISORC'07 : Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 393–402, Washington, DC, USA, 2007. IEEE Computer Society.

[130] Christine Morin, Jérôme Gallard, Yvon Jégou, and Pierre Riteau. Clouds, a New Playground for the XtreemOS Grid Operating System. Technical Report RR-6824, INRIA, February 2009.

[131] Monika Moser and Seif Haridi. Atomic Commitment in Transactional DHTs. In *Proceedings of the CoreGRID Symposium, Rennes, France*, page 151, 2007.

[132] Achour Mostéfaoui and Michel Raynal. Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors : A General Quorum-Based Approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, London, UK, 1999. Springer-Verlag.

[133] Athicha Muthitacharoen, Seth Gilbert, and Robert Morris. Etna : A Fault-tolerant Algorithm for Atomic Mutable DHT Data. Technical Report 993, MIT-LCS, June 2005.

[134] Matt W. Mutka and Miron Livny. Profiling Workstations' Available Capacity for Remote Execution. In *Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 529–544, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[135] Myrinet. http ://www.myri.com/myrinet/overview.

[136] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing (ICS '07)*, pages 23–32, New York, NY, USA, 2007. ACM.

[137] NorGrid. http ://www.norgrid.no.

[138] CoreGRID Network of Excellence. http ://www.coregrid.net/.

[139] Object Management Group (OMG). CORBA Component Model, v3.0. Technical Report formal/2002-06-65, Object Management Group, June 2002.

[140] OpenMosix. http ://openmosix.sourceforge.net.

[141] OpenPBS. http ://www.openpbs.org.

[142] OpenSSI. http ://openssi.org.

[143] OpenVZ. http ://wiki.openvz.org.

[144] Rob Pennington. Terascale clusters and the TeraGrid. In *Proceedings of 6th International Conference/Exhibition on High Performance Computing in Asia Pacific Region*, pages 407–413, Bangalore, India, December 2002. Invited talk.

[145] Christian Pérez, Thierry Priol, and André Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4) :417–429, 2003. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[146] Kenneth J. Perry and Sam Toueg. Distributed Agreement in the Presence of Processor and Communication Faults. *IEEE Transactions on Software Engineering*, 12(3) :477–482, 1986.

[147] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)*, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[148] S. L. Peterson and Phil Kearns. Rollback Based on Vector Time. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 68–77, 1993.

[149] Guillaume Pierre, Thorsten Schütt, Jörg Domaschka, and Massimo Coppola. Highly Available and Scalable Grid Services. In *Proceedings of the Third Workshop on Dependable Distributed Data Management (WDDM '09)*, pages 18–20, 2009.

[150] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt : Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings (TCON'95)*, pages 213–223, Berkeley, CA, USA, 1995. USENIX Association.

[151] Li Qi, Hai Jin, Ian Foster, and Jarek Gawor. HAND : Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4. In *PDP '07 : Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 155–162, Washington, DC, USA, 2007. IEEE Computer Society.

[152] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[153] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The Cost of Recovery in Message Logging Protocols. In *Symposium on Reliable Distributed Systems*, pages 10–18, 1998.

[154] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Sontent-Addressable Network. *SIGCOMM Computer Communication Review*, 31(4) :161–172, 2001.

[155] M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath, and D. Janakiram. Vishwa : A reconfigurable P2P middleware for Grid Computations. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*, pages 381–390, Washington, DC, USA, 2006. IEEE Computer Society.

[156] Ron I. Resnick. A Modern Taxonomy of High Availability. http ://www.generalconcepts.com/resources/reliability/resnick/, 1996.

[157] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[158] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing (PODC '91)*, pages 341–353, New York, NY, USA, 1991. ACM.

[159] Louis Rilling. Vigne : Towards a Self-Healing Grid Operating System. In *Proceedings of Euro-Par 2006*, volume 4128 of *Lecture Notes in Computer Science*, pages 437–447, Dresden, Germany, August 2006. Springer.

[160] Thomas Ropars, Emmanuel Jeanvoine, and Christine Morin. GAMoSe : An Accurate Monitoring Service for Grid Applications. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, Hagenberg Autriche, 07 2007.

[161] Thomas Ropars and Christine Morin. Fault Tolerance in Cluster Federations with O2P-CF. In *Resilience 2008, Workshop on Resiliency in High Performance Computing*, Lyon France, 05 2008.

[162] Thomas Ropars and Christine Morin. Active Optimistic Message Logging for Reliable Execution of MPI Applications. In *15th International Euro-Par Conference*, pages 615–626, Delft, The Netherlands, August 2009.

[163] Antony Rowstron and Peter Druschel. Pastry : Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In R. Guerraoui, editor, *Proceedings of International Middleware Conference (Middleware 2001)*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[164] Joseph F. Ruscio, Michael A. Heffner, and Srinidhi Varadarajan. DejaVu : Transparent User-Level Checkpointing, Migration and Recovery for Distributed Systems. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06)*, page 158, New York, NY, USA, 2006. ACM.

[165] SALOME. http ://www.salome-platform.org.

[166] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Paul Hargrove Jason Duell, and Eric Roman. The LAM/MPI Checkpoint/Restart Framework : System-Initiated Checkpointing. *International Journal of High Performance Computing Applications,*, 19(4) :479–493, 2005.

[167] André Schiper. Dynamic group communication. *Distributed Computing*, 18(5) :359–374, 2006.

[168] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach : a Tutorial. *ACM Computing Surveys*, 22(4) :299–319, 1990.

[169] Martin Schulz, Greg Bronevetsky, and Bronis R. Supinski. On the Performance of Transparent MPI Piggyback Messages. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 194–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[170] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A structured overlay for multi-dimensional range queries. In *Proceedings of Euro-Par 2007*, volume 4641 of *Lecture Notes in Computer Science*, pages 503–513, Dresden, Germany, 2007. Springer.

[171] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC : A Remote Procedure Call API for Grid Computing. In *Proceedings of the Third International Workshop on Grid Computing (GRID '02)*, pages 274–278, London, UK, 2002. Springer-Verlag.

[172] Mukesh Singhal and Ajay Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43(1) :47–52, 1992.

[173] A. Prasad Sistla and Jennifer L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing (PODC '89)*, pages 223–238, New York, NY, USA, 1989. ACM Press.

[174] TOP500 Supercomputing Sites. http ://www.top500.org.

[175] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data (SIGMOD '81)*, pages 133–142, New York, NY, USA, 1981. ACM.

[176] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6) :44–52, 1992.

[177] Sean W. Smith and David B. Johnson. Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollback. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, page 66, Washington, DC, USA, 1996. IEEE Computer Society.

[178] Sean W. Smith, David B. Johnson, and J. D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In *25th International Symposium on Fault Tolerant Computing Digest of Papers (FTCS-25)*, pages 361–371, Pasadena, California, 1995.

[179] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NetPIPE : A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[180] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback : a Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[181] Georg Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

[182] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord : A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1) :17–32, 2003.

[183] Nathan Stone, Derek Simmel, Thilo Kielmann, and Andre Merzky. An Architecture for Grid Checkpoint and Recovery Services. Technical Report GFD-I.93, Open Grid Forum, 2007.

[184] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computing Systems*, 3(3) :204–226, 1985.

[185] Vaidy Sunderam. PVM : A Framework for Parallel Distributed Computing. *Concurrency : Practice and Experience*, 2(4) :315–339, 1990.

[186] SweGrid. http ://www.swegrid.se/.

[187] Michal Szymaniak, Guillaume Pierre, Mariana Simons-Nikolova, and Maarten van Steen. Enabling Service Adaptability with Versatile Anycast. *Concurrency and Computation : Practice and Experience*, 19(13) :1837–1863, 2007.

[188] Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Versatile Anycasting with Mobile IPv6. In *AAA-IDEA '06 : Proceedings of the 2nd international workshop on Advanced architectures and algorithms for internet delivery and applications*, page 2, New York, NY, USA, 2006. ACM.

[189] Ben Temkow, Anne-Marie Bosneag, Xinjie Li, and Monica Brockmeyer. PaxonDHT : Achieving Consensus in Distributed Hash Tables. In *Proceedings of the International Symposium on Applications on Internet (SAINT '06)*, pages 236–244, Washington, DC, USA, 2006. IEEE Computer Society.

[190] Torque. http ://www.clusterresources.com.

[191] Peter Urban, Naohiro Hayashibara, Andre Schiper, and Takuya Katayama. Performance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems(SRDS '04)*, pages 4–17, Washington, DC, USA, 2004. IEEE Computer Society.

[192] Peter Urban and Andre Schiper. Comparing the Performance of Two Consensus Algorithms with Centralized and Decentralized Communication Schemes. Technical Report LSR-REPORT-2004-030, LSR, 2004.

[193] Peter Urban, Ilya Shnayderman, and Andre Schiper. Comparison of Failure Detectors and Group Membership : Performance Study of Two Atomic Broadcast Algorithms. *International Conference on Dependable Systems and Networks*, pages 645–655, 2003.

[194] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communication of the ACM*, 33(8) :103–111, 1990.

[195] S. Venkatesan, Tony Tong-Ying Juang, and Sridhar Alagar. Optimistic Crash Recovery without Changing Application Messages. *IEEE Transactions on Parallel and Distributed Systems*, 8(3) :263–271, 1997.

[196] Jim Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3) :5–7, Jul-Sep 1998.

[197] John Paul Walters and Vipin Chaudhary. Application-Level Checkpointing Techniques for Parallel Programs. *Distributed Computing and Internet Technology*, 4317 :221–234, 2006.

[198] Yi-Min Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4) :456–468, 1997.

[199] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :546–554, 1995.

[200] Aaron Weiss. Computing in the Clouds. *netWorker*, 11(4) :16–25, 2007.

[201] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. *20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pages 464–474, 2000.

[202] Uwe Wilhelm and Andre Schiper. A Hierarchy of Totally Ordered Multicasts. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems (SRDS '95)*, page 106, Washington, DC, USA, 1995. IEEE Computer Society.

[203] WMware. http ://www.wmware.com.

[204] XtreemOS. http ://www.xtreemos.eu.

[205] Xianan Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting. Fault-tolerant Grid Services Using Primary-Backup : Feasibility and Performance. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster '04)*, pages 105–114, Washington, DC, USA, 2004. IEEE Computer Society.

[206] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry : A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1) :41–53, January 2004.

# Services et protocoles pour l'exécution fiable d'applications distribuées dans les grilles de calcul


## Résumé

Une grille de calcul regroupe un très grand nombre de ressources de calcul hétérogènes, pouvant appartenir à différents domaines d'administration et distribuées géographiquement. Les grilles sont attractives car elles peuvent fournir à leurs utilisateurs les ressources nécessaires à l'exécution d'applications de calcul scientifique. Cependant, étant donné le nombre de ressources pouvant composer une grille, les risques de défaillances matérielles y sont élevés, compromettant la terminaison des applications soumises par les utilisateurs.

Cette thèse vise à assurer l'exécution fiable d'applications distribuées dans les grilles de calcul, solutions adaptées à la taille, la volatilité, et l'hétérogénéité de celles-ci. Pour répondre à ces enjeux, nous apportons trois principales contributions.

XtreemGCP est un service de recouvrement arrière conçu dans le cadre du projet européen XtreemOS. Il assure le redémarrage automatique des applications défaillantes en utilisant des mécanismes de recouvrement arrière transparents pour les applications. XtreemGCP est conçu de manière générique pour pouvoir y intégrer simplement différentes solutions de recouvrement arrière, et ainsi pouvoir mieux répondre aux besoins des applications.

Semias assure la haute disponibilité et l'auto-réparation de services de grille tels que XtreemGCP en fournissant la première mise en œuvre de techniques de duplication active dans un réseau logique structuré. Les propriétés d'auto-réparation de Semias, mises en évidence expérimentalement sur la plate-forme Grid'5000, assurent la disponibilité des services dans un environnement très dynamique.

O2P est un protocole de recouvrement arrière, mis en œuvre dans la bibliothèque Open MPI, fondé sur l'enregistrement de messages optimiste actif. Exploitant une gestion distribuée de l'enregistrement des messages, il offre une solution de recouvrement arrière passant à l'échelle pour les applications à échange de messages. Ses propriétés ont été démontrées expérimentalement sur Grid'5000.



## Mots clés

Grille de calcul, tolérance aux fautes, haute disponibilité, auto-réparation, passage à l'échelle, recouvrement arrière, duplication active.